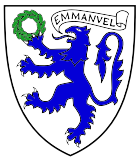
UNIVERSITY OF
CAMBRIDGE

Complete spatial safety for C and C++ using CHERI capabilities

Alexander Leslie Richardson



Emmanuel College

October 2019

This dissertation is submitted for the degree of Doctor of Philosophy

Abstract

Lack of memory safety in commonly used systems-level languages such as C and C++ results in a constant stream of new exploitable software vulnerabilities and exploit techniques. Many exploit mitigations have been proposed and deployed over the years, yet none address the root issue: lack of memory safety. Most C and C++ implementations assume a memory model based on a linear array of bytes rather than an object-centric view. Whilst more efficient on contemporary CPU architectures, linear addresses cannot encode the target object, thus permitting memory errors such as spatial safety violations (ignoring the bounds of an object). One promising mechanism to provide memory safety is CHERI (Capability Hardware Enhanced RISC Instructions), which extends existing processor architectures with *capabilities* that provide hardware-enforced checks for all accesses and can be used to prevent spatial memory violations. This dissertation prototypes and evaluates a *pure-capability* programming model (using CHERI capabilities for *all* pointers) to provide *complete* spatial memory protection for traditionally unsafe languages.

As the first step towards memory safety, all language-visible pointers can be implemented as capabilities. I analyse the programmer-visible impact of this change and refine the pure-capability programming model to provide strong source-level compatibility with existing code. Second, to provide robust spatial safety, language-invisible pointers (mostly arising from program linkage) such as those used for functions calls and global variable accesses must also be protected. In doing so, I highlight trade-offs between performance and privilege minimization for implicit and programmer-visible pointers. Finally, I present *CheriSH*, a novel and highly compatible technique that protects against buffer overflows between fields of the same object, hereby ensuring that the CHERI spatial memory protection is *complete*.

I find that the byte-granular spatial safety provided by CHERI pure-capability code is not only stronger than most other approaches, but also incurs almost negligible performance overheads in common cases (0.1% geometric mean) and a worst-case overhead of only 23.3% compared to the insecure MIPS baseline. Moreover, I show that the pure-capability programming model provides near-complete source-level compatibility with existing programs. I evaluate this based on porting large widely used open-source applications such as PostgreSQL and WebKit with only minimal changes: fewer than 0.1% of source lines.

I conclude that pure-capability CHERI C/C++ is an eminently viable programming environment offering strong memory protection, good source-level compatibility and low performance overheads.

Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except as declared in the Preface and specified in the text. It is not substantially the same as any that I have submitted, or am concurrently submitting, for a degree or diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. I further state that no substantial part of my dissertation has already been submitted, or is being concurrently submitted, for any such degree, diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. This dissertation does not exceed the degree-committee-approved limit of 66 000 words.

Alexander Leslie Richardson
15th October 2019

Acknowledgements

I would like to take this opportunity to thank the following people for their invaluable help and contribution to this project without whom this research would have not been possible:

First of all, I would like to thank my supervisor, Robert Watson, for giving me the opportunity to work on the immensely interesting CHERI project. I cannot thank him enough for the guidance and help he has provided over the past years since I started my MPhil in 2014. I am very grateful for his encouragement and useful critiques of this research work as well as his willingness to answer all the questions I had.

I would like to thank David Chisnall for teaching me about the internal workings of the CHERI compiler at the start of my PhD. I would also like to extend my thanks to Khilan Gudka for all his work on the tools that I relied on for my PhD (and for my MPhil before), in particular for making WebKit and libc++ run on CHERI.

I would like to thank Nathaniel Filardo for his extremely helpful and detailed feedback on hundreds of pages of my dissertation draft as well as providing assistance with his LaTeX and TikZ skills. Thank you also for the many useful discussions around new CHERI ideas, some of which have made it into this dissertation.

I would also like to thank the wider CHERI project members, including Simon Moore, Brooks Davis, Lawrence Esswood, John Baldwin, Alfredo Mazzinghi, Edward Napierała, Jessica Clarke, Alexandre Joannou and Jonathan Woodruff. The CHERI project is a truly collaborative effort that would not be possible without the specific skill-set brought by each individual member. I have learnt so much from every one of you.

I would furthermore like to thank my examiners, Alan Mycroft and Howard Shrobe for their invaluable feedback and suggested corrections.

I could not have undertaken this research without the funding provided by Hewlett-Packard Enterprise and DARPA. I would like to thank Hewlett-Packard Labs, and particularly Dejan Milojicic, for the internship in Palo Alto that sparked many further research ideas.

I would like to thank my family for their constant support and for giving me the opportunity to study and do research at Cambridge. Thank you also for proofreading my entire dissertation (multiple times!).

Finally, I would like to thank Claire for her patience and amazing support over the past four years. Thank you also for many hours proofreading of my thesis. I could not have done this without you!

Contents

1 INTRODUCTION

Even after decades of research into mitigation techniques and safer programming languages, C- and C++-language memory-safety errors remain the root cause for many widely exploited software vulnerabilities, causing vast financial damage every year. In the UK alone, the estimated cost of cybercrime in 2016 was £26bn [107]. A recent example of this is the WannaCry ransomware that spread world-wide, infecting hundreds of thousands of systems and resulting in substantial losses: it cost the British National Health Service £92m [56] and the total world-wide damage was estimated to be up to $8bn [123]. This malware exploited a bug, Common Vulnerabilities and Exposures (CVE) 2017-0144 [168], where the underlying issue was a memory-safety problem (a buffer overflow caused by integer overflow) [90] in the Windows SMBv1 file sharing protocol.

This is not an outlier: in 2019, Microsoft reported that around 70% of all security issues in their products relate to memory safety [153]. While the exploit primitives have changed over time [153, slide 13] (most likely due to mitigation techniques being enabled by default), out-of-bounds memory accesses (spatial memory errors) are still listed as the number one root cause for Microsoft CVEs. Similarly, Google reports that 44% of all Android Linux Kernel flaws discovered between January 2014 and April 2016 were due to missing or incorrect bounds checking [235]. Additionally, many of the other listed errors such as `NULL` pointer dereferences, use-after-free or integer overflows require a spatial memory violation for exploitation. Overall this shows that somewhere between 56% and 73% of all exploitable Android Linux kernel vulnerabilities in recent years could be addressed by having a C implementation that provided strong spatial memory and pointer integrity guarantees. Another Google presentation from October 2018 states that over 50% of *High/Critical* security bugs in Chrome and Android are memory-safety issues [206, slide 3]. A more recent presentation by the same author states that between May 2017 and May 2018 over 60% of all reported CVEs were memory-safety issues [203, slide 10]. It also shows that between July 2017 and July 2018 [203, slide 14] almost 1000 out-of-bounds accesses and 500 use-after-free errors were found in Google's internal data-centre software. This is very similar for bugs found by Google's Project Zero [69] (over 60% memory-safety issues [203, slide 16]) and Mozilla CVEs (also over 60% [203, slide 17]). Finally, the Android developer documentation states that 'as of 2016, about 86% of all vulnerabilities on Android are memory safety related' [8] and the '*2019 CWE Top 25 Most Dangerous Software Errors*' report lists buffer handling errors as the most serious vulnerability class [45].

Due to the prevalence of memory-safety errors, many operating-system vendors have started deploying vulnerability mitigations by default and have enabled them by default in their compilers. Probabilistic exploit mitigation techniques such as address-space layout

randomization (ASLR) [225] have been enabled by most operating systems for many years. Another example, stack canaries [46], are enabled by default in the Apple [49] and Microsoft compilers [149] and are used for Red Hat Enterprise Linux and Fedora packages [250]. While these techniques come at a fairly small overhead (e.g. 10% worst-case overhead for stack canaries [50]), more expensive mitigations have been deployed recently. One such example is Control-flow Integrity (CFI) [1]: the Android kernel for Google's devices is built with Clang CFI [232] and Microsoft has enabled CFI for the Windows kernel and applications since 2015 [19]. The LLVM [134] CFI implementation can incur whole-program slowdowns of 9% for some SPEC benchmarks [28], although this number will be higher for programs with more indirect calls. Another example is kernel page-table isolation [92] or *retpolines* [233], software mitigations for recently disclosed speculative execution vulnerabilities (which often speculatively violate spatial safety) [121, 141]. These mitigations can be expensive — especially in virtualized environments[1] — yet have been deployed and are enabled by default in most operating-system kernels [31, 91, 151]. Hardware vendors have also started to include vulnerability-mitigation techniques in their latest instruction-set architecture (ISA) revision. For example, ARM v8.5a includes a feature called Branch Target Indicators (BTI) [86] that can enforce a weak form of forward-edge (i.e. function call) CFI. Intel has added a similar feature, Indirect Branch Tracking, as part of their Control-flow Enforcement Technology (CET) [108] extensions. The CET extensions also include a hardware implementation of shadow stacks [237], which can be used to protect backward-edge control flow (i.e. function returns).

One solution to the memory-safety problem would be rewriting all existing code in safer programming languages. However, considering the millions of lines of existing C and C++ that would need to be rewritten, this is unlikely to be a viable strategy. Recompiling legacy code in a mostly backwards-compatible, safer dialect of C that allows incremental addition of memory safety sounds like an attractive option, yet prior attempts at memory-safe annotated C dialects have required changes to anywhere between 10% and 35% of all lines of code [67, 116] and are therefore almost as invasive as a complete rewrite in a safe language. Another option to tackle the security vulnerabilities would be adding memory safety to C and C++. However, current state-of-the art software implementations of spatial memory protection result in performance overheads of anywhere between 60% and over 100% [6, 65, 215, 221]. Despite being a recent hardware implementation of bounded pointers, Intel Memory Protection Extensions (MPX) [110] still incurs around 50% overhead for SPEC2006 [174] as it uses disjoint bounds metadata and inserts additional bounds checking instructions in the generated code. By design, it requires additional memory accesses for every bounds check, is not thread-safe and has a fail-open policy [174, 256]. Therefore, MPX would be ineffective and inefficient as a vulnerability-mitigation tool, and the feature has in fact now been withdrawn from GCC [76] and the Linux kernel [154].

[1] After updating the FreeBSD build servers that we use for continuous integration to a kernel with these mitigations, the build duration more than *quadrupled* for some jobs.

An underlying problem is that the memory model of the C programming language is generally that of a linear array of bytes. We seek to replace that model with a semantically richer one that we refer to as the *object-memory* model. [2] In this model, memory is viewed as an unordered collection of *objects*. Objects are characterized by three properties:

Identity: Given two references to objects it is possible to tell if they refer to the same or different objects.

Extent: An Object occupies a fixed amount of memory. It is possible to tell whether a reference is within that bounded area of memory.

Type: An object has a type that dictates the operations that are meaningful to perform on it. For example it is not legal to add a number to a string but it is legal to extract a character from the string.

Crucially, ordering is not a property of objects; telling whether two objects are adjacent is not a meaningful operation. A system to adheres to this model is considered to be memory safe. Our challenge is that these properties (although consistent with the C standard) are not adhered to in many existing code bases. Moreover, implementing such a model on contemporary hardware would incur significant perfomance costs.

The performance problems of memory-safe C could be addressed by building upon hardware support that facilitates enforcement of these properties. A promising foundation for this is CHERI [40, 41, 54, 74, 117, 244, 246, 247, 248, 254, 256, 258, 259], an instruction-set extension that provides memory capabilities on top of existing ISAs such as MIPS, RISC-V or ARMv8. Prior CHERI research has primarily focused on selective use of CHERI capabilities (e.g. in small sandboxes). In contrast, this dissertation explores and evaluates the use of CHERI for all pointers in C/C++-language codebases. CHERI has been designed to allow language-level pointers (e.g. in C or C++) to be implemented as CHERI memory capabilities. Corruption of pointers or array indices is a highly effective exploitation strategy, and most memory corruption attacks rely on overwriting pointers or reinterpreting data as pointers, thus violating the identity and type safety properties [221]. Therefore, a complete C and C++ runtime environment that uses CHERI capabilities instead of integer values as pointers should be able to mitigate almost all memory-safety attacks (see Section 2.2). Moreover, CHERI capabilities contain all the required bounds information in the pointer and therefore do not require any table lookups for bounds and permission checks. This property ensures that CHERI has a much lower performance overhead than other techniques. In fact, CHERI might even improve performance compared to the current state since most other mitigation techniques are unnecessary in an environment that uses capabilities for all pointers.

Another benefit of CHERI is strong architectural primitives to support compartmentalization [170, 246, 247, 248]. For example, 85% of Android kernel vulnerabilities are in vendor-supplied kernel drivers [235], which suggests that some degree of compart-

[2] This is not to be confused with the more elaborate models used by object-oriented programming languages.

mentalization could thus reduce the impact of vulnerabilities. CHERI's architectural compartmentalization support could be an ideal solution to this problem — especially when combined with appropriate static analysis [95, 96]. CHERI memory safety is an essential foundation for CHERI compartmentalization [247, 248].

The UK government's Industrial Strategy Challenge Fund has recognized the problems caused by lack of memory safety and CHERI's potential in addressing these. Together with various private companies, it will provide funding [55] to produce an experimental CPU and board that integrates CHERI into the ARMv8-A ISA [240] by 2021 [234]. ARM believes that 'CHERI [...] compartmentalization can make future systems inherently more robust against known attacks and so-called Zero Day attacks we may face in future' [88]. To deploy compartmentalization on top of CHERI, a usable pure-capability programming model (see Chapter 3) and a linkage model that can provide compartmentalization (see Chapter 4) must exist first. This thesis addresses essential foundations that are required for such a secure-by-design system.

1.1 Contributions

In this dissertation, I propose, prototype and evaluate a series of architectural, language, compiler and toolchain techniques supporting complete spatial memory safety using CHERI capabilities for the C and C++ programming environments:

- I demonstrate that all source-code-visible C/C++-language pointer types can be implemented using CHERI capabilities. To improve compatibility, I propose changes to existing CHERI C semantics (including an *address* interpretation of capabilities) that significantly reduce lines-of-code modifications compared to prior efforts. Additionally, I extend the CHERI ISA and compiler to enable efficient code generation for the new semantics.
- I demonstrate that all implicit pointers in the C/C++-language runtime, including those implementing global variables, control flow and thread-local storage can be implemented using CHERI capabilities, for both statically and dynamically linked applications. I compare various linkage models, including those closest to current linkage (in which implied pointers are reached via the program counter) and those more suitable for future CHERI-based compartmentalization schemes (in which data is reached via a separate set of capabilities, thus allowing multiple instances of data to share common code).
- I prototype and evaluate new architectural features (*sentry* capabilities) that can be used to provide low-overhead isolation between individual dynamic shared objects (DSOs) without relying on CHERI object types. Building upon these new features, I propose and implement a novel mechanism to enforce correct global variable accesses in C and C++, *Global Visibility Enforcement (GVE)*, and demonstrate that it can be implemented purely in the linker without changing the compiler-generated code.

- I demonstrate that CHERI can be used to enforce bounds at a sub-object granularity for existing C programs while retaining almost complete source-code compatibility. I further show that this new extension to pure-capability CHERI C/C++ semantics, *CHERI sub-object hardening (CheriSH)*, supports modern C/C++ code as well as code written prior to modern C standards (such as the FreeBSD system libraries and even the kernel). Moreover, I evaluate performance and spatial protection properties and show that pure-capability code with CheriSH provides *complete* spatial safety for C and C++ code.
- I present a classification of idioms that are incompatible with CHERI C semantics (either with or without sub-object bounds) and showcase new compiler warnings that can detect many of these issues statically rather than at run time.
- Finally, I demonstrate that the performance overhead of C and C++ implemented using CHERI capabilities for all pointers is acceptable for real-world deployment. I benchmark on a field-programmable gate array (FPGA) using a realistic 5-stage pipeline and microarchitecture comparable to ARM7TDMI.

Throughout, I evaluate with respect to source-level compatibility, performance and protection. I use a broad corpus of open-source code — including the FreeBSD operating system, PostgreSQL database, libFuzzer fuzzing tool, and WebKit web-rendering framework — to analyse the source-level compatibility impact of pure-capability CHERI. I use a range of off-the-shelf benchmarks including SPEC and MiBench to explore how different techniques perform, and also provide root-cause performance analysis for CHERI's overheads in real-world software stacks. I demonstrate low overall overheads in most workloads and develop new architectural and compiler techniques to improve CHERI performance. I use a range of approaches to evaluate protection, including the Juliet and BOdiagsuite buffer-overflow test suites.

1.2 Publications

Over the course of my PhD I co-authored various publications, many of which directly contribute to this dissertation.

1.2.1 Publications directly contributing to this dissertation

Peer-reviewed conference papers

- '*CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment*' by Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey

Son and Jonathan Woodruff [54]. In this paper on full operating-system support for CHERI pure-capability programming, my refinements to pure-capability C/C++ and performance optimizations (see Chapter 3) as well as the implementation of CHERI dynamic linkage (see Chapter 4) were essential contributions. This publication received a **best paper award at ASPLOS 2019**.

- '*Exploring C Semantics and Pointer Provenance*' by Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson and Peter Sewell [147]. For this paper on pointer-provenance semantics in the C programming language, I analysed the real-world occurrences of certain idioms using pure-capability C. This analysis directly contributes towards Section 3.4.2.

Technical reports

- '*Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 7)*' by Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, David Chisnall, Brooks Davis, Nathaniel Wesley Filardo, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alex Richardson, Peter Rugg, Peter Sewell, Stacey Son and Hongyan Xia [246] is the primary reference and specification of the CHERI architecture. Throughout this dissertation I introduce various new architectural features, which are specified in this technical report.
- I also contributed to the extended technical report version of '*CheriABI*' [54], which includes more implementation details: '*CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-Time Environment*' by Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son and Jonathan Woodruff [53].

Peer-reviewed extended abstract and presentation

- '*Secure Linking in the CheriBSD Operating System*' by Alexander Richardson and Robert N. M. Watson [188] showcases parts of my work on pure-capability linkage (see Chapter 4).
- '*Protecting C++ Applications Using CHERI*' by Khilan Gudka, Alexander Richardson and Robert N. M. Watson [94] presents some challenges and opportunities for pure-capability C++ (see Section 3.7). It also includes some detail on porting WebKit to pure-capability C++ (see Section 6.1.3.9)

1.2.2 Other publications

- '*Efficient Tagged Memory*' by Alexandre Joannou, Jonathan Woodruff, Robert Kovacsics, Simon W. Moore, Alex Bradbury, Hongyan Xia, Robert N. M. Watson, David Chisnall, Michael Roe, Brooks Davis, Edward Napierala, John Baldwin, Khilan Gudka, Peter G. Neumann, Alfredo Mazzinghi, Alex Richardson, Stacey Son and A Theodore Markettos [117].
- '*CheriRTOS: A Capability Model for Embedded Devices*' by Hongyan Xia, Jonathan Woodruff, Hadrien Barral, Lawrence Esswood, A. Joannou, Robert Kovacsics, David Chisnall, Micheal Roe, Brooks Davis, Edward Napierala, John Baldwin, Khilan Gudka, Peter G. Neumann, Alexander Richardson, Simon W. Moore and Robert N. M. Watson [259].
- '*CHERIvoke: Characterising Pointer Revocation Using CHERI Capabilities for Temporal Memory Safety*' by Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel W. Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G. Neumann, Simon W. Moore, Robert N. M. Watson and Timothy M. Jones [258].
- '*Cornucopia: Temporal Safety for CHERI Heaps*' by Nathaniel Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, Theo Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann and Robert Watson [74].
- '*Separating Translation from Protection in Address Spaces with Dynamic Remapping*' by Reto Achermann, Chris Dalton, Paolo Faraboschi, Moritz Hoffmann, Dejan Milojicic, Geoffrey Ndu, Alexander Richardson, Timothy Roscoe, Adrian L. Shaw and Robert N. M. Watson [3].
- '*Memory-Side Protection With a Capability Enforcement Co-Processor*' by Leonid Azriel, Lukas Humbel, Reto Achermann, Alex Richardson, Moritz Hoffmann, Avi Mendelson, Timothy Roscoe, Robert N. M. Watson, Paolo Faraboschi and Dejan Milojicic [17].
- Extended technical report version of '*SOAAP*' [95] with more implementation details: '*Clean Application Compartmentalization with SOAAP (Extended Version)*' by Khilan Gudka, Robert N. M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann and Alex Richardson [96].

1.3 Work performed in collaboration with others

The CHERI project is enormously complex, with changes the entire hardware-software stack. Pure-capability C/C++ has benefited from many person-years of work on hardware design, toolchain and operating-system support, as well as testing infrastructure. Completing this

dissertation would not have been possible without the contribution of the past and present project members, who have laid the foundations for this work over the past nine years. I carried out all work detailed in this thesis (unless explicitly stated otherwise) except for the following:

- David Chisnall wrote the majority of LLVM/Clang support for pure-capability CHERI C until April 2018, when I took over as the compiler maintainer. This work was the baseline against which my exploration of performance, compatibility, linkage, sub-object bounds, etc. was performed. His contribution also includes various CHERI compatibility warnings, apart from the new warnings listed in Chapter 3 which I implemented. Additionally, I implemented the compiler support for pure-capability dynamic linking and sub-object bounds.
- Brooks Davis performed most of the work in CheriABI support for CheriBSD (both userspace and kernel-side). In this co-design work, I worked with Brooks to identify, classify and address compatibility concerns (see Chapter 3). I also contributed to CheriBSD by making various compatibility fixes, enabling CheriBSD to build on Linux/macOS and adding support for dynamic linking and CheriSH.
- Khilan Gudka ported QtWebKit to work as a pure-capability binary and did the initial port of libc++. He also made some of the compiler changes to support C++ and added the `__cheri_addr` and `__cheri_offset` casts.
- Jessica Clarke implemented an initial CHERI dynamic linkage prototype and modified thread-local storage for pure-capability code to work without ambient capabilities. Her work influenced my design choices for pure-capability linkage (see Chapter 4).
- Alfredo Mazzinghi ported the CheriBSD kernel to work as a pure-capability binary and made the necessary changes to run it with CheriSH enabled. The pure-capability CheriBSD kernel is one of the case studies in my compatibility evaluations.
- Edward Tomasz Napierała ported NGINX to work on top of CheriBSD and assisted me in porting PostgreSQL. Both projects are used in the compatibility evaluation.
- John Baldwin made C++ exception handling work without ambient capabilities and added CHERI support to GDB. This work would have been significantly more difficult and taken much longer without a functioning debugger.
- Jonathan Woodruff added the new instructions I propose in this dissertation to the CHERI FPGA, allowing me to benchmark on a realistic micro-architecture.

1.4 Outline

This dissertation explores the hypothesis that CHERI capabilities can be used for all explicit and implicit pointers in a C or C++ language runtime with a high degree of compatibility and low performance overheads. Chapter 2 provides the background for the remainder of this dissertation. In Chapter 3, I explore differences between programmers' expectations and pure-capability CHERI C/C++. I also propose and prototype pure-

capability C/C++ improvements addressing these problems. Chapter 4 focuses on the linkage of programs and the implications on privilege minimization of implicit pointers. Chapter 5 presents CheriSH, a technique to enforce complete spatial memory safety. Finally, I evaluate performance and compatibility of pure-capability C/C++ in Chapter 6 and summarize my findings in Chapter 7.

BACKGROUND 2

In this chapter I first give an overview of current attacks exploiting lack of memory safety in C/C++ applications. Next, I introduce CHERI as the background for Chapter 3 and Chapter 5, and explain how CHERI can address memory-safety issues. Finally, I give a short overview over the process of linking and loading as this is relevant to Chapter 4.

2.1 Memory-corruption attacks

To understand the benefits that CHERI provides in terms of preventing memory corruption attacks, we must first explore the currently prevalent attack vectors. C/C++ suffer from lack of spatial memory safety: missing bounds checks on language-visible buffers/pointers may allow attackers to manipulate language-invisible code pointers (e.g. return addresses) to execute arbitrary (malicious) code. This has led to many critical vulnerabilities but so far no solution has been deployed [45, 153, 221, 235].

While usage of memory-safe languages is increasing, a large (and growing) corpus of memory-unsafe C/C++ code still exists. Memory safe(r) variants of C have been proposed in the past (e.g. '*Cyclone*' [116], '*CCured*' [162] and '*Checked C*' [67]), yet all change the language and thus require non-trivial changes to real-world source code. Therefore, the current systems deploy numerous (mostly probabilistic) vulnerability-mitigation techniques in the hope of preventing exploits.

2.1.1 Code-injection attacks

The possibility to inject malicious code and manipulate control flow via buffer handling errors has been known since at least 1972 [7, App. I] and was first widely exploited by the Morris Worm [66, 199, 217]. Prior to vulnerability-mitigation features being enabled by default, it was possible for an attacker to overflow buffers on the stack, fill them with code of their choosing, and then overwrite the return address to execute the injected code [177].

Stack canaries One commonly deployed (probabilistic) vulnerability-mitigation technique is stack canaries (also known as stack-protector). This technique places a fixed value (the canary) between the top of the stack frame and the return address. On function return, it checks the canary value for modification and if so aborts execution [46]. Stack canaries prevent most contiguous stack buffer overflows (assuming the canary value has not been leaked) at a moderate cost of up to 10.5% overhead [50] that can be reduced by selectively instrumenting functions [43].

This mitigation approach is enabled by default on most operating systems [49, 149, 250], and is probably one of the underlying reasons that more recent exploits rely on targeted out-of-bounds accesses instead of contiguous overflows [153, slide 18].

Non-executable memory An early (deterministic) mitigation for code injection was to make stacks non-executable [57]. However, this still allowed attackers to inject code on the heap. Therefore, most operating systems choose to architecturally specify memory management unit (MMU) permissions that prevent pages being writable and executable at the same time [226]. This technique is also known as W⊕X or Data Execution Prevention (DEP) [150]. To bypass this protection, attackers can resort to using just-in-time (JIT) compilers (e.g. JavaScript runtimes in web browsers) to generate code sequences that can be used maliciously (e.g. by jumping to the middle of x86 instructions) [22].

2.1.2 Code-reuse attacks

As vulnerability-mitigation features such as W⊕X/DEP make it harder for attackers to write arbitrary code sequences to executable memory, many newer attacks manipulate control flow to reuse existing code instead. Instead of calling an existing function, the attacker will often jump into the middle of existing functions to manipulate the stack and return pointers. These sequences of instructions are referred to as *gadgets*, and chaining sequences of them to execute arbitrary code is called return-oriented programming (ROP) [191, 207]. In these kinds of attacks, injected pointers almost always point to the language-invisible pointers (return addresses, internal function addresses, etc.) rather than to a language-visible attacker-controlled buffer. While the original attack relied on the x86 `ret` instruction, it is also possible without using this instruction [36]. ROP also works on RISC architectures such as SPARC [26]. Similar attacks using jumps (jump-oriented programming) [23], loops (loop-oriented programming) [133], C++ *vtables* (counterfeit object-oriented programming) [198] or `sigreturn()` [25] have since been invented.

Recent research into code-reuse attacks has shown that it is possible to automatically create ROP exploits without knowledge of the target binary against a `fork()`-ing server [20]. This kind of attack, blind return-oriented programming (BROP), has also been shown to work against secure enclaves such as Software Guard Extensions (SGX) where code and data are invisible to an attacker [136].

Address-space layout randomization The most widely deployed mitigation feature against code-reuse attacks is address-space layout randomization (ASLR) [225]. ASLR is a defence mechanism that randomizes the addresses of code and data, thereby reducing the probability that an attacker can guess the address of a usable code sequence (gadget) for the exploit. However, this mechanism can be bypassed if the attack can extract a pointer value within a given region, since only the start addresses are randomized and offsets

within that region are still constant. Therefore, finer-grained randomization techniques have been developed, but these can also be bypassed [214].

Recent research has shown that ASLR is fundamentally insecure as randomization can be bypassed using timing side-channels on MMU page table walks. Importantly, this attack succeeds even when executed from JavaScript without access to precise timers (e.g. in web pages being rendered by a browser) [85].

Control-flow Integrity Control-flow Integrity (CFI) is a security property proposed by Abadi et al. [1]. If a program maintains CFI, then any given program execution adheres to a pre-computed valid control-flow graph (CFG). If enforced dynamically, CFI rules out ROP attacks, as these corrupt control flow to new attacker-created execution graphs.

Many techniques to enforce this property have been proposed and implemented with varying degrees of protection vs. performance trade-offs [1, 2, 5, 34, 86, 108, 222]. Implementations can be either coarse-grained (i.e. allowing transitions to *any* valid jump target, even if this transition is not valid according to the CFG) or fine-grained (i.e. checking that the transition exists in the CFG). Different CFI implementations protect either forward-edges (i.e. indirect jumps), backward-edges (i.e. returns) or both.

Many operating-system vendors ship CFI mechanisms by default. Microsoft ships a coarse-grained CFI mechanism, Control Flow Guard [222], and has enabled CFI for the kernel and Edge Browser [19]. Android also ships with LLVM-based CFI for the kernel [232] and userspace [144]. Additionally, instructions to enforce coarse-grained CFI are being added to both Intel (Control-flow Enforcement Technology [108]) and ARM's (Branch Target Indicators [86]) coming architectures.

While CFI increases the difficulty for attackers, coarse-grained variants are not a guaranteed defence [52]. Another problem with CFI techniques is that they rely on a statically computed CFG, and must therefore over-approximate or risk valid programs failing at run time. A recent approach, per-input CFI (πCFI) [169] constrains valid jump targets based on concrete inputs instead of the statically computed CFG. However, Evans et al. have shown that even fine-grained CFI can be bypassed by a sufficiently capable attacker, as most programs contain call-sites where corruption of function arguments can result in targeted code execution [71].

Code-pointer Integrity Code-pointer Integrity (CPI) is an approach to prevent control-flow hijacking by providing memory safety for code pointers (but not for other pointers) [221]. This technique was first implemented by Kuznetsov et al. (spatial-only protection in the current implementation) [131]. Using static analysis, it is possible to protect only pointers that are considered 'sensitive'. All sensitive pointers (and pointers to sensitive values) are stored in a separate memory region and all accesses of these pointers are checked. This approach reduces the overhead compared to full software-enforced memory safety. Full CPI incurs performance overheads of up to 44.2% for some SPEC benchmarks.

This overhead can be reduced to 17.2% by using the weaker Code-pointer Separation (CPS) model, which only protects direct code pointers and omits some checks. An even lower overhead model (but with much weaker protection guarantees), SafeStack, uses a different stack for return addresses, register spills and escaped variables. This is a low-overhead (0–4% according to [131]) information-hiding scheme that obscures the location of some attractive targets for code-reuse attacks and is available in Clang with the flag `-fsanitize=safe-stack`. However, it relies on ASLR to hide the location of return pointers and can therefore be bypassed by reducing the entropy available for the safe stack (e.g. by spawning threads [84]).

Evans et al. presented an attack that can bypass CPI without causing any crashes [70]. The creators of CPI argue that this is an attack on a specific implementation of CPI and not a fundamental weakness [130]. However, on x86-64 and ARM this technique relies on information hiding: the location of the secure memory region must not leak to an attacker. CPI therefore remains a probabilistic defence rather than a deterministic one and is insufficient to defend against an attacker with arbitrary code-execution. Additionally, Veen et al. have recently shown that code-reuse attacks are possible in the presence of both CPI and CFI [236].

2.1.3 Data-only attacks

Some attacks do not rely on manipulating control-flow data (such as return addresses) but instead overwrite function arguments or variables that change control flow [37]. Automated ways of creating such attacks have been proposed [103, 114], and Hu et al. have shown that these kinds of attacks are Turing-complete [104]. However, these exploits generally rely on out-of-bounds accesses so can be prevented by techniques that enforce spatial safety. Other defences against this kind of attack include data-flow integrity [33] and many other hardware- and software-based mitigations [4, 39, 171, 197].

2.1.4 Information leakage

Sometimes an attacker may be able to obtain information that should be inaccessible (e.g. private keys, passwords and so on). Moreover, many current vulnerability-mitigation techniques rely on information hiding (e.g. by randomizing attacker-relevant addresses). However, this information can often be obtained by attackers in many ways. For example, side-channels may allow attackers to infer values, or pointer values might be included in (debug) logs.

Modern CPUs speculatively execute instructions in advance (and discard the result if it ends up being unnecessary) to improve performance. However, this often results in observable side effects that can be exploited by attackers to obtain information. Recent examples include '*Meltdown*' [141] and the '*Spectre*' [121] attacks. Mitigating these cases of information leakage can be expensive as doing so prevents CPUs from using these

performance-enhancing features. Nevertheless, this kind of leakage is considered very severe and therefore most operating-system vendors apply software workarounds such as kernel page-table isolation (KPTI) [91, 92] or *retpolines* [151, 233].

2.1.5 Multi-purpose mitigation techniques

The mitigation techniques listed in the previous subsections share one common downside: they are specific to one exploit technique, and do not provide protection against multiple classes of vulnerabilities.

Compartmentalization, sandboxing and fault isolation Instead of preventing attacks, it is also possible to limit the impact of a successful exploit by running high-risk code in a different domain with lower privileges (e.g. separate processes, virtual machines, etc.) [10, 34, 146, 200, 238, 239, 241, 262]. This ensures that a successful attack is still constrained by the limited permissions of the current domain and requires another exploit to escape the restricted execution environment. These domains can contain anything from managed language runtimes to entire virtualized operating systems. Compartmentalization can be expensive due to additional domain transitions, but architectural support can make this overhead manageable. For example, CHERI provides the architectural primitives for low-cost provably secure compartmentalization [167] within the same address space [247, 248]. As the topic of this dissertation is memory protection rather than compartmentalization, I only tangentially consider this aspect of CHERI in the following chapters.

Bounds checking and spatial memory safety As most attacks rely on memory corruption, another solution to the problem is to ensure that pointers cannot be used out-of-bounds, i.e. enforcing *spatial memory safety* [221]. Memory safety and bounds-checking tools associate bounds metadata either with the object/allocation or the pointer. Using per-pointer metadata allows enforcing sub-object bounds whereas using per-object metadata does not. I present the memory-safety techniques that can protect sub-objects in more detail in Section 5.6.

Most approaches insert bounds checks on pointer arithmetic or use, which can be enforced by either hardware [58, 110, 132, 157, 158] or software [6, 63, 64, 65, 99, 119, 128, 129, 160, 166, 172, 173, 209, 260, 263]. Another approach is blacklisting, in which memory between objects is made inaccessible. This can be done e.g. by allocating one object per-page with inaccessible pages in-between [179], using software checks [201] or with hardware modifications [196, 202, 205, 211]. One problem with blacklisting is that capable attackers can jump over the region and access valid memory [181, 253].

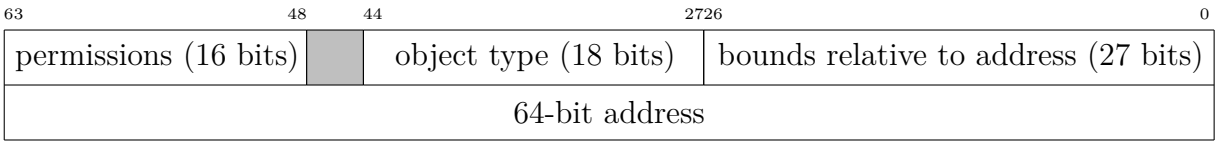


Figure 2.1: CHERI capability representation with a 64-bit address [246, 254]. The tag (validity) bit is stored out-of-band.

2.2 CHERI

CHERI (Capability Hardware Enhanced RISC Instructions) is an instruction-set architecture (ISA) extension that provides architectural *capabilities* for conventional CPU architectures. CHERI draws ideas from prior capability-based computer systems [73, 138], but unlike most other systems is a *hybrid* capability system that allows for incremental adoption rather than requiring complete software re-engineering. CHERI adds tagged memory to distinguish capabilities from data. Each architectural capability contains an address, a lower and an upper bound, permissions and an object type (see Figure 2.1).

CHERI capabilities are most commonly used to reference memory addresses and can replace machine-level (and language-level) pointers.[1] The architecturally defined properties of CHERI allow them to enforce *spatial* and *referential* safety and include features that facilitate implementing *temporal* safety. These properties are highlighted in Figure 2.2 and are as follows:

Bounds and permissions CHERI capabilities include a lower and an upper bound, which allows the hardware to prevent any access outside these bounds. Additionally, permissions ensure that e.g. a read-only pointer cannot be used to store data. These two properties guarantee *spatial safety*.

Monotonicity All operations on CHERI capabilities are monotonic, i.e. they can only reduce rights but never increase them. This ensures that capabilities that have been bounded to a region can never be modified to grant accesses outside the region. Similarly, permissions cannot be added to a capability once removed.

Integrity and provenance validity CHERI capabilities include a single-bit *validity tag* (stored out-of-band [117]) to precisely distinguish capabilities from raw data. Most instructions operating on capability registers (e.g. loads and stores) require this bit to be set and fault otherwise. Tags are used to enforce *pointer integrity* (which we refer to as *referential safety*) since it is no longer possible to create valid pointers from data (as is commonly done in code-reuse attacks). Additionally, tag bits make it possible to precisely locate valid pointers, which is beneficial for *temporal* safety [74, 258]. CHERI capabilities can only be created by using instructions that explicitly derive from another valid capability (*provenance validity*).

[1] They can also be used to implement opaque tokens of authority that are software-interpreted.

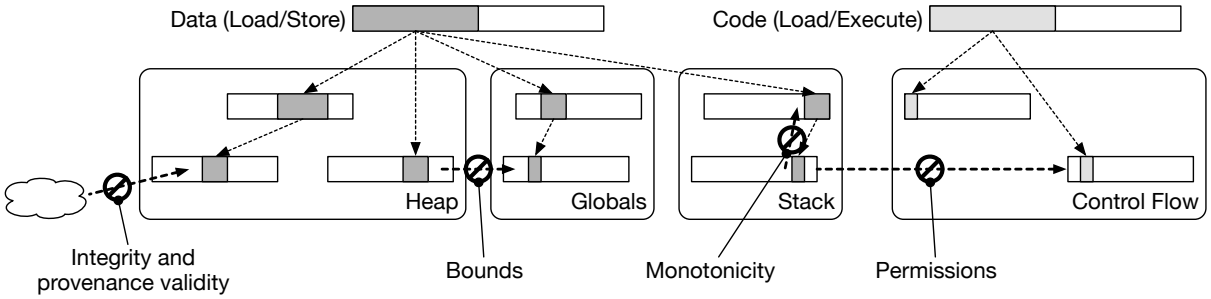


Figure 2.2: CHERI enforces protection semantics for pointers.[2]

Sealing and object types CHERI capabilities also include an *object type*. If the type is not equal to `-1`, any further modification or dereferencing results in hardware traps (i.e. the capability is *sealed*). Sealed capabilities can be used either as software-defined immutable tokens of authority or to implement compartmentalization (using special instructions for non-monotonic transitioning between compartments). However, each isolated compartment requires one object type and only a limited space of object types exists (just under 18 bits in the current implementation). Therefore, this dissertation introduces a variant of sealed capabilities, *sentry* capabilities, that do not use object types in Section 4.7.1.

While the current implementation of CHERI extends the MIPS ISA, these properties are portable across architectures and prototypes for RISC-V and ARMv8 [13] are in development.

CHERI's architectural capabilities can be used in numerous ways. Examples include the topic of this dissertation, complete spatial memory safety using pure-capability compilation (e.g. for the entire FreeBSD userspace [54]), but also scalable compartmentalization [247, 248], capability-aware embedded real-time operating systems [259], sandboxing of native code in managed languages [40], temporal safety using sweeping revocation [74, 258] and many more future applications.

CHERI capabilities encode all required information inside the capability itself. This property allows CHERI to avoid lookup tables or other kinds of indirection and therefore the potential for vastly better performance than other approaches for spatial safety (see Section 6.2.9). Originally, CHERI capabilities were 256 bits wide [256], but in the current architecture redundancy between the address, lower and upper bound is exploited to compress all CHERI capabilities to 128 bits [117].[3] Reducing the size of capabilities is not only beneficial for performance (due to reduced memory overheads), but also improves compatibility by reducing pointer alignment requirements. It does however have implications that software must be aware of, some of which are discussed in Sections 3.4 and 5.3.4. For the remainder of this dissertation CHERI capabilities are assumed to be 128-bits unless stated otherwise.

[2]Figure drawn from '*An Introduction to CHERI*' [244].

[3]Additionally, a 64-bit capability format with a 32-bit virtual address exists [259].

For further details on CHERI, I recommend reading either '*An Introduction to CHERI*' for a higher-level overview [244], or for more detailed information the full architecture specification, which includes design choices and rationale [246].

CHERI as a vulnerability-mitigation technique Probabilistic buffer-overflow mitigations such as stack-protectors or ASLR make exploits *more difficult* but do not completely prevent them. Moreover, many techniques can be bypassed by information leakage or suffer from small secret sizes that can be brute-forced. In contrast, CHERI is deterministic: it is impossible to bypass CHERI's defences rather that having a reduced chance of success. A pure-capability CHERI environment would prevent spatial safety violations (e.g. out-of-bounds writes) or referential safety violations (e.g. creating a code pointer from raw data), and therefore can mitigate the attacks listed in Section 2.1. The following chapters explore the foundations for such an environment, for language-visible pointers (see Chapter 7) as well as invisible, linkage-derived pointers (see Chapter 4).

Nevertheless, some kinds of attacks are possible even with CHERI. As previously described, pure-capability C/C++ does not protect sub-objects, so data-only attacks that corrupt adjacent structure members are a potential attacker vector. As this is a realistic threat, I introduce a defence that protects sub-objects, CheriSH, in Chapter 5. For this dissertation temporal safety is out of scope (see [74, 258] for a possible CHERI-based approach), however, most (current) attacks — even those exploiting temporal safety violations (e.g. double-free or use-after-free) — rely on being able to exploit at least one spatial safety violation or being able to forge a pointer before being able to execute the actual payload [153, slide 18]. Additionally, logic flaws or type confusions could result in incorrect arguments being passed to a function (e.g. calling `execve("/bin/sh")`). Often this confusion will be caused by spatial or temporal violations (which can be addressed by CHERI), but programming logic errors are also explicitly out-of-scope. Finally, information leakage via side channels is not generally addressed by CHERI, but some cases can be prevented by adding micro-architectural support for compartments [249].

2.3 Linkage

In conventional software design, a linker has the conceptually simple task of combining multiple input files into a single runnable program image and resolving all external references to valid pointers. However, many subtleties exist in this often overlooked part of program execution. This process is especially important for pure-capability C/C++ since it is the origin of most language-invisible pointers (e.g. function call targets or global variables), which must have minimal permissions and be correctly bounded so that pure-capability CHERI can enforce spatial safety (see Chapter 4). In this section I only introduce aspects that are required in the context of this dissertation. For further detail I

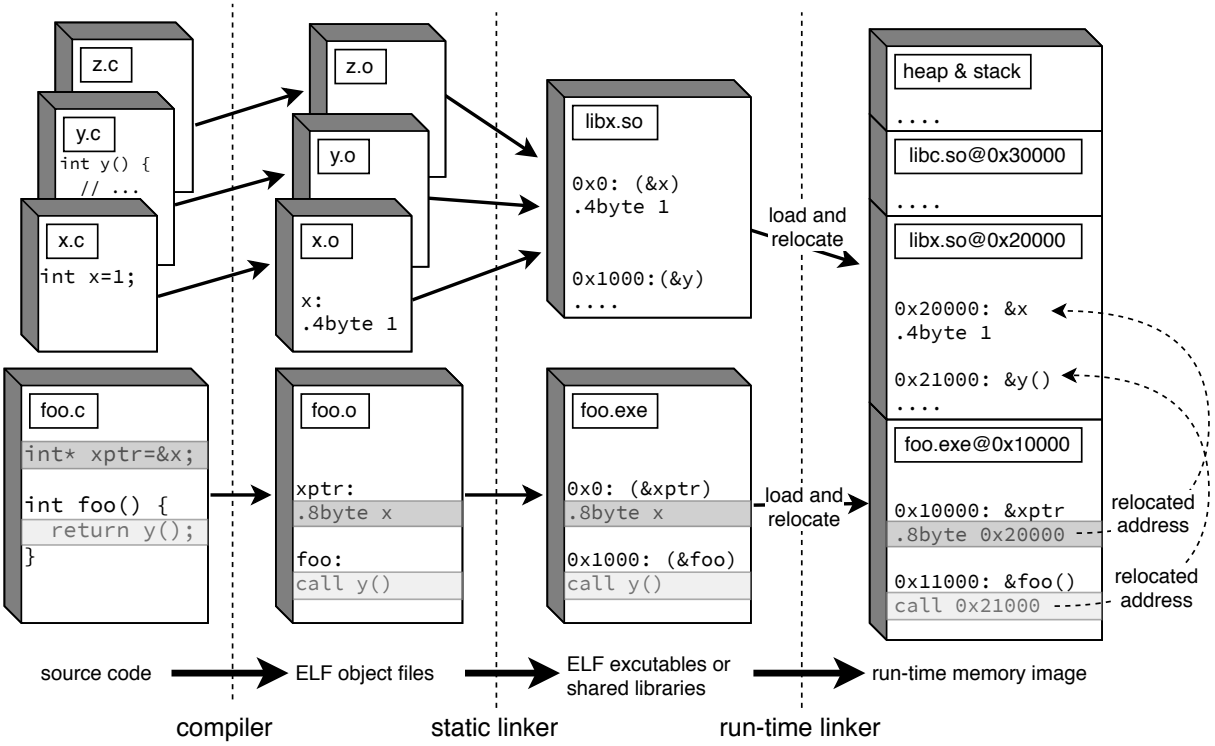


Figure 2.3: Overview of the different steps for linking and loading an executable.

recommend reading '*Linkers and Loaders*' [137]. While this book is dated, it does contain most information required to understand current systems.

Historically, most systems included a single link editor (`ld`) that combines input object files and a very simple loader that loads the generated output into memory. This approach can still be used for *static* linking, where all symbol references in input object files are resolved and the linker creates a single binary file that can be directly executed when mapped to an appropriate address by the *loader* (usually this is the operating-system kernel). On current systems dynamic linking is common, in which case some external references remain unresolved and the final stage of combining inputs happens at run time. To enable run-time reference resolving the compiler must generate code that can be relocated to any run-time address (so-called position-independent code). This final step of linking and relocating is performed by the run-time linker (RTLD). While many executable binary formats exist [137, chapter 3], this dissertation assumes that all binaries use the Executable and Linkable Format (ELF) [120].

2.3.1 Relocation processing

When the compiler or linker generates ELF files, some values cannot be resolved until later with these unresolved values being *relocations* [224]. For example, often only the offset to the dynamic shared object (DSO) base address, but not the run-time address, is

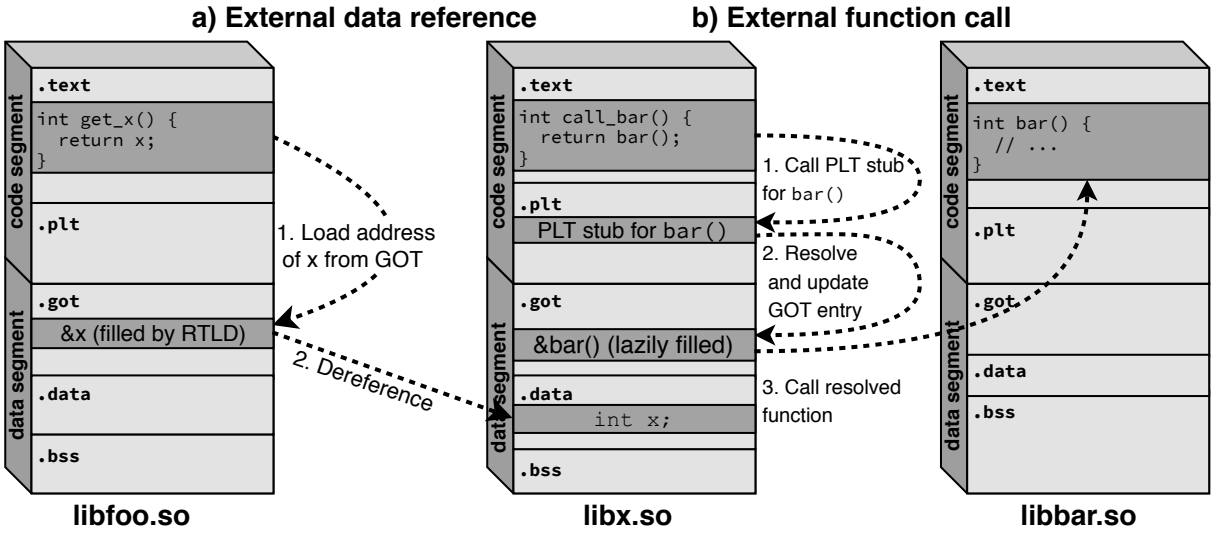


Figure 2.4: Using the GOT and PLT to access external symbols.

known at static link time. In this case the static linker must emit dynamic relocations so that the run-time linker can write the final pointer value [137, chapters 7&8].

To initialize global pointers, architectures using the ELF file format use a section in the file, .rel.dyn or .rela.dyn, that contains an array of `Elf_Rel` or `Elf_Rela` structures. These structures include information on the location of the value to be initialized (`r_offset`, an offset relative to the start of the DSO), information on how to relocate the value (`r_info`, an integer that contains information about the target symbol and the type of relocation) and optionally, in the case of `Elf_Rela` an addend (an integer that should be added to the resolved value of the relocation before writing it to memory). To give one example, position independent executables (or shared libraries) on an x86 system will contain many `R_X86_64_RELATIVE` relocations that instruct RTLD to relocate the target by the DSO base address. These relative relocations are used to initialize pointers to objects within the current DSO. External references to symbols in other DSOs use the same mechanism, but with a different relocation type (`r_info`) that instructs RTLD to perform symbol lookups instead of simply adding the base address.[4]

2.3.2 GOT and PLT

When linking statically, the addresses of function call targets and global variables are known and can often be computed relative to registers such as the program counter. However, when dynamically linking many of these values are not known until run time. Therefore, most architectures reference external symbols via the so-called global offset table (GOT), a table that the dynamic linker fills with the resolved addresses.[5] External function calls

[4]The static linker may be able to simplify some relocations: For example, if an input object file contains an external reference to a symbol that happens to be linked into the same DSO, this external reference can be replaced with a relative one.

[5]The GOT can be found relative to the program counter as this is always a known fixed offset.

generally use the procedure linkage table (PLT), which is a series of short code sequences (one per external reference) that transfer control to the actual target function [137, chapter 10] (see Figure 2.4). The PLT also enables on-demand resolution of the target function (see Section 4.3.3) which can speed up program loading for large applications noticeably. In my implementation of pure-capability CHERI linkage, I use a similar approach for the PLT, but without resolving the target via the GOT (see Section 4.3).

3 PURE-CAPABILITY CHERI C/C++

This chapter introduces pure-capability C/C++, a C/C++ implementation in which *all* pointers (both language-visible and implicit ones) are represented using CHERI capabilities, thereby providing spatial and referential memory safety (see Section 2.2) for these presently unsafe languages. Pure-capability C/C++ breaks some assumptions commonly made by C programmers relating to pointer representation (see Sections 3.2 and 3.4), calling conventions (see Section 3.3), behaviour of casts and pointer arithmetic (see Sections 3.5 and 3.6) or optimizations (see Section 3.9). Although these assumptions often hold for common architectures using integer pointers, many are contrary to the C standard's definitions and do not necessarily apply to all integer-pointer architectures.[1]

Over the past four years, the CHERI project has refined the pure-capability C/C++ model to better support code making these kinds of assumptions without impacting the protection provided by CHERI. However, there are no perfect solutions in this space: as we increase protection, we risk breaking current software, impacting compatibility. We see this challenge primarily in low-level parts of system software that themselves implement virtual memory, memory allocators, linkers and so on. As we shift into more language-visible territory (integer-pointer properties, sub-object bounds), we see more challenging trade-offs. Here we are guided by pragmatism, weighing up frequency of occurrence (and the ease of adjusting these to be compatible with pure-capability C/C++) against the ability to provide a clean and consistent language model with the strongest protection guarantees.

Developing a full pure-capability C/C++ environment has been a large collaborative effort that could not have been undertaken by one person. My key contributions in this effort that are highlighted in this dissertation include developing, prototyping, and evaluating:

- performance work in the compiler and instruction set (resulting in near-identical performance to integer-pointer MIPS, see Section 6.2),
- refinements to improve source-level compatibility (such as introducing an *address* interpretation of capabilities, see Section 3.6) resulting in up to 60% fewer required changes (see Section 6.1),
- significant functional improvements to the pure-capability programming model (including introducing sub-object bounds, see Chapter 5),
- creating multiple pure-capability linkage models (see Chapter 4) and

[1] For example, most x86 and ARM CPUs support unaligned data accesses, but other processors may not.

- improved compiler static checking for unavoidable areas of change — to discover problems at compile time instead of requiring dynamic run-time debugging of a crashing or malfunctioning program.

In order to enable the transition from initial experiments with small benchmarks to supporting a huge code corpus (tens of millions of lines of code), I performed detailed analysis and made significant improvements to our build and testing approaches.

These contributions were essential to '*CheriABI*' [53, 54] and all other papers we have published that build upon pure-capability C/C++ [74, 117, 254, 258, 259]. The analysis for this chapter also contributes to the internal report '*An Introduction to Pure-Capability CHERI C/C++ Programming*' [243], and the upcoming next version of the '*CHERI Programmer's Guide*' [242]. Additionally, my analysis of out-of-bounds pointers in Section 3.4.2 was included in '*Exploring C Semantics and Pointer Provenance*' [147].

This chapter structured as follows: Sections 3.1 to 3.4 introduce the pure-capability programming model and highlight some key differences between this model and C on a traditional architecture. It is important to note that there are many different possible implementations of pure-capability C/C++. While a naıve pure-capability model may adhere to the ISO C standard [112], it may not be able to run any real-world C code, since that frequently relies on additional assumptions. Therefore, the next part of this chapter (Sections 3.5 and 3.6) focuses on initial design choices (mostly related to type conversions) that resulted in incompatibilities with real-world C as well as the resulting changes I made to pure-capability semantics to improve source-level compatibility. Section 3.7 lists the changes required to extend the pure-capability model from C to C++. If we consider real-world deployability, performance of pure-capability C/C++ is equally as important as compatibility aspects. Therefore, I also provide a brief overview of changes I made to the CHERI ISA and toolchain to enable performant execution of pure-capability code in Section 3.8. The source-level compatibility and performance of pure-capability C/C++ is evaluated in Chapter 6 after Chapters 4 and 5 introduce the remaining requirements to enable complete spatial memory safety on top of CHERI.

3.1 The pure-capability programming model

Initially, CHERI-based C only supported a mode where every pointer that should be a capability had to be explicitly annotated with `__capability` [256]. This model is referred to as the *hybrid* model as it allows C language-level pointers to be implemented either as an integer address or a CHERI capability. In this mode, the existing C runtime is largely unchanged and only annotated pointers behave differently. A C implementation where pointers are implemented using CHERI capabilities (i.e. the *pure-capability* C environment) was proposed in 2015 in '*Beyond the PDP-11*' [41]. This paper also explored the interaction of CHERI capabilities with certain C idioms and concluded that using capabilities for

pointers is a feasible approach. However, until recently we did not have a full operating system running pure-capability code [54] and therefore could not validate this hypothesis.

At the start of my PhD in 2015, David Chisnall had recently added compiler support for the initial experiments with pure-capability CHERI. However, at the time we only supported running small (statically linked) sandboxes,[2] and most code still used the hybrid mode. Pure-capability code also derived most language-invisible pointers from virtual addresses relative to two ambient capabilities (with bounds spanning the entire address space): the default data capability (`$ddc`) for data references and the program counter capability (`$pcc`) for code. The initial pure-capability C/C++ design avoided exposing virtual addresses to enable certain temporal safety techniques such as copying garbage collection. As we applied this model of C and C++ to more and more code, we discovered that although the initial model was a workable foundation, the design choice to avoid virtual address leakage resulted in high compatibility costs. Therefore, we departed from being garbage-collection friendly as one of the design goals, and instead focused on providing a spatially safe C implementation that is highly compatible with C programmers' expectations [147, 148] and are looking at sweeping revocation instead of garbage collection for temporal safety [74, 258]. Over the course of the past four years and many person-years of effort, we have since added full operating-system support for pure-capability binaries, using a new process application binary interface (ABI), *CheriABI*, with full POSIX compatibility (including dynamic linking, see Chapter 4) and the ability to run complex applications such as WebKit or PostgreSQL [53, 54]. My contributions in this effort include the following:

Source-compatible protection for explicit pointers Although most exploits target language-invisible pointers to subvert control flow, they often originate from improper use of language-visible pointers (e.g. lack of bounds checking). Implementing all pointers as CHERI capabilities can address the most severe cause of errors: lack of spatial memory safety [45]. However, programmers often make (sometimes incorrect) assumptions about the properties of pointers and existing software can break when they change. This chapter explores these semantic differences caused by changing the properties of language-visible pointers.

In general, pure-capability C/C++ code should look and behave identically to C/C++ code on contemporary architectures. However, programmers should be aware of some differences, which I highlight in the following sections. Most sections in this chapter talk about C specifically, but most also apply equally to C++ programs.[3] Nevertheless, we did have to make some C++ specific changes to the language and compiler, which are listed

[2]The fact that pure-capability C/C++ originally ran in sandboxes was encoded in the compiler flags until March 2017: passing `-mabi=sandbox` enabled pure-capability compilation, but this has since been renamed to `-mabi=purecap`.

[3]Many C-compatibility issues are less common in C++, or are abstracted in a class, which usually means that fewer changes are required when using C++ as the implementation language.

in Section 3.7. Some problems described in this chapter were caused by initial design choices such as pointer-comparison semantics or attempting to support garbage collection. Based on these findings, we have since adjusted the default semantics for pure-capability code, and substantially reduced the amount of changes required (see Section 6.1 for a compatibility evaluation).

Protection for implicit pointers For full protection, not only visible pointers (e.g. the ones that would have been annotated with `__capability` in the hybrid ABI) must be implemented as capabilities, but more importantly also pointers that are hidden from the programmer and only visible to the compiler/runtime (e.g. return addresses, function call targets, global variables or C++ *vtables*). Many memory-safety techniques focus either on language-visible pointers or implicit pointers. In contrast, pure-capability CHERI is able to protect *all* pointers in C/C++ programs, providing non-bypassable, deterministic spatial memory protection.[4] Importantly, CHERI capabilities are unforgeable and provide monotonically decreasable permissions [167]. This property provides strong integrity guarantees for data and control flow and prevents commonly used type confusion between raw data and pointers, which is the underlying cause for many existing exploits.

Protection for sub-objects Prior work on CHERI has mostly focused on spatial memory safety at the granularity of memory allocations. However, for certain kinds of exploits (e.g. data-only attacks, see Section 2.1.3) this protection is insufficient. This problem can be addressed by further narrowing bounds to a sub-object level. Yet, this results in compatibility and performance trade-offs, which are detailed and evaluated in Chapter 5.

Once all these aspects have been explored, I evaluate overall performance and compatibility in Chapter 6.

3.2 Pointer representation

The most obvious change between pure-capability C/C++ and conventional implementations is the different representation of pointers. In the pure-capability compilation mode, all language-visible pointers (and invisible ones such as return addresses) are CHERI capabilities. This can result in portability problems that will typically be found due to hardware exceptions at run time, but for some cases we can issue diagnostics during compilation.

[4] We could also envision a model in which only the invisible pointers use capabilities (especially return addresses, as this makes ROP much more difficult); however, we have not explored this model in practice. A compiler flag exists that enables a mode where only return addresses are capabilities and all other pointers are still integers. However, in this mode code still runs with an ambient default data capability (`$ddc`), so a sufficiently powerful attacker could derive any valid return capability from `$ddc`.

To support integer-to-pointer casts, pure-capability C/C++ defines `uintptr_t` as a capability type, `__uintcap_t`, which is 128 bits with a tag. However, the *integer range* of this type is only 64 bits. Integer-to-pointer casts and `uintptr_t` arithmetic is one difference that caused larger compatibility problems. As they are (unfortunately) common in C code, this is a recurring theme in the following sections.

3.2.1 Pointer size

On 64-bit systems, CHERI capabilities have a width of 128 bits, but the arithmetic properties of a 64-bit integer. Yet, implicit knowledge of pointer size is embedded in many projects and must be addressed when porting to CHERI.

Hard-coded pointer size Some projects assume that pointers are either 32 or 64 bits wide and therefore omit code for handling other cases. These missing conditional cases resulted in the code for 32-bit systems being used for pure-capability CHERI. We had to change this in multiple programs such as PostgreSQL or LLVM's compiler-rt library (see Section 6.1). This pattern is difficult to detect statically but will often result in linker errors due to missing functions or obvious errors at run time.

Changed structure sizes Structure layouts are often optimized for 64-bit pointers, which can result in excessive padding when compiling for CHERI. To detect structures that should be reordered, David Chisnall added a `-Wexcess-padding` warning. Padding is usually purely an efficiency problem, yet sometimes increased structure sizes can cause run-time failures. We found that e.g. NGINX assumed maximum sizes for some structures and used this knowledge when sizing buffers. We therefore had to adjust allocation sizes when porting NGINX to run as a pure-capability binary.

`uintptr_t` Some code assumes that a pointer will fit into the largest integer type (e.g. `uint64_t` or `long`).[5] This is not true when using capabilities and casts to integer and back will strip the capability metadata. The correct type to use for this purpose is `uintptr_t`: 'any valid pointer to void can be converted to this type, then converted back to pointer to void, and the result will compare equal to the original pointer' [113, §7.20.1.4]. Arguably, any code converting pointers and integers should be using `uintptr_t`, but even the C2x standard does not guarantee that this type is available [113, §7.20.1.4]. Additionally, the standard does not guarantee that performing arithmetic on `uintptr_t` before casting it back to a pointer is supported. For pure-capability C/C++, we support this idiom (see Section 3.6), but we may strip tags if the result is too far out-of-bounds (see Section 3.4).

[5] Another common assumption is that a `double` can hold pointer values. Some dynamic language runtimes use *NaN-boxing* (or *NaN-packing*) [93] and rely upon this.

`ptrdiff_t` The type `ptrdiff_t` 'is the signed integer type of the result of subtracting two pointers' [113, §7.19.2]. However, some programmers assume that `ptrdiff_t` (and often also `size_t`) are the same size as a pointer and can be used interchangeably.[6] For example, the Qt framework defines its `qintptr` type as an alias for `qptrdiff`. However, for pure-capability code `ptrdiff_t` is the size of a virtual address and untagged—whereas `intptr_t` must be a capability to retain tag bits and capability metadata when casting.

`vaddr_t` Often programmers wish to obtain the virtual address of a pointer. For pure-capability C/C++, it would be inefficient to use `uintptr_t` (128 bits), so we introduced an unsigned equivalent of `ptrdiff_t` for CheriBSD, the virtual address type `vaddr_t`. This type was also required to fix problems with the *offset*-centric view of the original pure-capability model (see Section 3.6 for details).

Integer range Although CHERI capabilities are twice the size of conventional pointers, the addressable range remains 64 bits. This difference between integer size and valid range caused many internal assertions in the Clang compiler. In some (rare) cases this also led to build failures for C code that assumes the largest integer that can be stored in `uintptr_t` is $2^{\texttt{sizeof(void*)*8}}\texttt{-1}$. We also discovered the assumption that `-1` converted to `uintptr_t` results in an all-ones value in snmalloc [139], yet for pure-capability code it results in a capability with an all-ones address but zeroes in the remaining bits.

In-memory order Another source of surprising behaviour is the in-memory order of capabilities. Currently, both 128-bit and 256-bit CHERI store the capability address as the second 64-bit value in memory. Therefore, storing a pointer and loading 64 bits from that address will not yield the address bits but the metadata. This property complicated the porting effort for WebKit where some generated code assumed that reading back a stored pointer as an integer would read the address. The upcoming RISC-V implementation of CHERI will use a little-endian memory order and place the address first.

3.2.2 Bounded pointers

Another obvious difference to a contemporary C implementation is that pointers now include (monotonically reducible) bounds information and permissions, and can therefore enforce complete spatial safety. Most programmers need not be aware of these bounds, as many different components (such as compiler, operating system (OS) kernel, dynamic linker, memory allocator, etc.) narrow bounds and remove permissions to minimize the chance of spatial violations (see '*CheriABI*' [53, 54] for more detail). For example, the compiler bounds stack allocations (see Section 3.8.2), and the dynamic linker bounds global

[6] This is not only true for legacy code. For example, a `static_assert` that checks that the size of `size_t` is the same as `void*` was added to Qt as recently as 2017 .

variables and function pointers (see Chapter 4). In one case programmers should explicitly set bounds: when writing a memory (sub-)allocator. Omitting the bounds-setting will not cause compatibility problems; however, all pointers handed out by the allocator will permit access to adjacent allocations.

In most cases, having bounds on pointers does not result in any compatibility issues but instead finds real problems in code that contains out-of-bounds accesses. However, bounded pointers do break the idioms of obtaining a pointer to a different object by adding an offset (see Section 3.9.2) or updating of pointers after `realloc()` (see Section 3.9.3). Although one of the goals of pure-capability C/C++ is to maximize compatibility, the changes to make such code compatible with pure-capability C/C++ are small and both idioms rely on undefined behaviour. Therefore, we believe that not supporting these idioms is acceptable.

Pure-capability code can also be compiled with support for bounding sub-objects, which is detailed in Chapter 5. In this case, additional C idioms no longer due to the narrower bounds, so there is a higher compatibility cost. Therefore, sub-object bounds are not currently enforced by default, but we may choose to do so in the future.

3.2.3 Alignment of capabilities

Due to the nature of tagged memory and the CHERI implementation using one tag per capability sized region in DRAM, capabilities have strict alignment requirements. CHERI capabilities can only be loaded and stored at a 16-byte (32-byte for CHERI-256) alignment boundary. Any attempt to load or store capabilities at an insufficiently aligned address will cause a run-time trap. Some architectures have the same restriction for 2/4/8-byte memory accesses and will trap if these are not correctly aligned. However, userspace programs will usually not be terminated due to these unaligned loads as the kernel can emulate them and pretend that the trap never happened. This is completely transparent to the user, and programs with unaligned loads/stores still function correctly (albeit with potentially reduced performance). For CHERI capabilities it is impossible to emulate the loads and stores because no tag bit is associated with the under-aligned address.

When the compiler allocates space for a structure type, it will ensure that the whole structure is aligned such that all the fields will be placed at an address that satisfies the alignment requirements of that type. However, we found cases where these alignment guarantees can be violated.

Packed structures When programmers require precise control over the layout of structures, they mark them as *packed*, which instructs the compiler not to insert any padding. Therefore, care must be taken to ensure that fields are sufficiently aligned to be accessed.[7]

[7]Unfortunately, most code targets architectures that support unaligned accesses of pointers and therefore does not consider this a problem — unless the performance degradation is significant.

The alignment of fields can be adjusted using the `_Alignas` alignment specifier or compiler-specific attributes (see Appendix D for more details), but most code assumes 32/64-bit pointers. For example, marking `struct { long l; void* ptr; }` *packed* will result in run-time crashes as the field `ptr` is only aligned to `sizeof(long)`. We therefore added a compiler warning whenever we encounter a capability (i.e. a pointer or `uintptr_t`) in a packed structure that is not aligned to a multiple of the capability size.

Normally, the compiler adds padding at the end of structures to ensure that they can safely be used in an array. This padding is omitted for *packed* structures. This property may cause some members to no longer be laid out correctly when the structure is used in an array. For example, `struct { void* p; int i; }` will result in a 20-byte structure, which means the next element will start at a four-byte alignment boundary. As this could also result in run-time crashes, we added a compiler diagnostic for this case that suggests setting the alignment requirement for the entire structure to `sizeof(void*)`.

`max_align_t` The type `max_align_t` is defined in C as 'an object type whose alignment is the greatest fundamental alignment' [113, §7.19.2] and in C++ as a 'type whose alignment requirement is at least as great as that of every scalar type' [111, §21.2.4p5]. This type is sometimes used to align allocations, but when running tests for libc++ we discovered that the definition was incorrect for pure-capability code. We had to fix the type definitions to be sufficiently aligned for CHERI capabilities as the implementation assumed that `long double` was the scalar type with the largest alignment requirement.

Pointers obtained from custom allocators The C standard guarantees that pointers returned from `malloc()` are sufficiently aligned to point to any 'object with a fundamental alignment requirement' [113, §7.22.3], i.e. at least the alignment of `max_align_t`.[8]. This is true for the `malloc()` implementation in CheriBSD, since we adjusted it for pure-capability CHERI [53]. However, we noticed that some projects (e.g. PostgreSQL or NGINX) include custom allocators that assume that aligning values to either 16 bytes or `sizeof(long double)` is sufficient to store any scalar type. We also found a similar issue in SQLite, where the allocator wraps `malloc()` and stores 8 bytes of metadata before returning the pointer, thus reducing the alignment to 8. These problems result in a run-time alignment exception and are difficult to detect statically. However, they will generally cause a crash early in the program execution and should therefore be found easily.

Impact of compressed capabilities Alignment errors have become less common since we have transitioned from using a 256-bit capability representation to a 128-bit compressed model [254] by default. Many allocators already align to multiples of 16 bytes, so these will

[8] Any code that requires stricter alignment must use application programming interfaces (APIs) such as `posix_memalign()` or C11 `aligned_alloc()`

work out-of-the-box in pure-capability mode. However, compressed capabilities impose additional constraints on alignment, which are detailed in Section 3.4.3.

3.2.4 Preserving tags when copying memory

A slightly more subtle case where CHERI alignment requirements can cause incompatibilities happens when copying data. Capability tag bits are only preserved when using capability-sized loads and stores (`CLC`/`CSC`). We found two cases where capability-bearing data is copied using loads and stores of a smaller size, thus stripping the tag bits.

Under-aligned copy destination The example in Listing 3.1 will sometimes—in cases where the stack allocation is not sufficiently aligned—result in `buffer` not containing the tag bit from `value`. If `buffer` happens to be aligned to `alignof(void*)` at run time, `memcpy()` will copy the tag bits since an appropriate memory operation will be used.[9]

```
extern uintcap_t cap;
char buffer[sizeof(uintcap_t)];
// buffer may not be aligned to 16
memcpy(buffer, &cap, sizeof(cap));
do_something(buffer);
```

Listing 3.1: Possibly tag-stripping under-aligned `memcpy()`.

However, if `buffer` is only aligned to four or eight bytes, then `memcpy()` will end up using integer memory access instructions and produce a byte-by-byte identical copy of the capability but without the tag bit set. One example for this problem was in CheriBSD when using the `setjmp()`/`longjmp()` `jmp_buf` data structure. Due to an incorrect structure declaration the saved register state would lose tag bits 50% of the time due to being misaligned and resulted in crashes when hitting *CTRL+C* in `csh`. As misaligned copies result in difficult to debug run-time failures far from the source of the problem, I modified the `memcpy()` and `memmove()` slow-path to `abort()` if the source contains valid capabilities but the destination is under-aligned.[10] If this is the intended behaviour, the programmer should explicitly use a non-tag-preserving copy.

Inlining of `memcpy()` A common (and performance-critical) compiler optimization is inlining of `memcpy()` and `memmove()`. These functions will be expanded inline at the call site by the compiler if the size of the copy is known and the expansion is deemed faster than a function call. It will then use copy instructions based on the minimum alignment of the source and destination. If the compiler assumes a 4-byte aligned pointer, it will use integer load/store instructions (`CLW` and `CSD`) instructions to copy the data. At run-time, if the pointer is aligned to 16 bytes, it should have used capability-size accesses (`CLC` and `CSC`) to copy instead. For all other architectures but CHERI, using a four-byte

[9] It is to be noted that `memcpy()` must always use `CSC`/`CLC` to copy data that is at least capability-aligned which means all `memcpy()` implementations must be CHERI-aware.

[10] This diagnostic from `memcpy()` has resulted in some unexpected problems. Originally, I made this a fatal error that would terminate the program. However, we found multiple cases where it triggered due to copying uninitialized (or partially initialized) data that sometimes contained tags. We therefore changed the fatal error to a warning that can be made fatal by setting a `sysctl` variable.

copy instead of a larger copy is just a performance issue. However, for CHERI this is a correctness problem since a copy of a capability using non-capability-sized loads and stores will copy all the representation bits, but not the tag bit, thereby yielding an invalid capability. To avoid surprising tag losses, the MIPS backend will emit a `memcpy()` library call for any insufficiently aligned copy larger than the capability size. This is not ideal for performance, but currently required for correctness.

Compiler diagnostics Due to these problems, I added a new warning that triggers when `memmove()` or `memcpy()` (and the `__builtin` variants) are called with a source containing a capability but without the destination being known to be aligned to at least capability size. The compiler cannot know the run-time alignment of the pointer and estimates based on the type of the pointer. Initially, I implemented this diagnostic in the Clang frontend. However, the C frontend only sees a very conservative estimate for the alignment, which makes this diagnostic difficult to implement without false positives.

As the required information is only available in the backend, I added two new attributes to LLVM's memory transfer intrinsics. The first is *must-preserve-cheri-tags* that indicates that a given copy must retain tag bits. When encountered, the compiler will always emit a `memcpy()` if it cannot expand the loop in a tag-preserving way (e.g. due to alignment constraints). The second is *frontend-memtransfer-type* that is populated by the C/C++ frontend so that the backend can emit a diagnostic containing the type name. For example:

```
warning: memcpy operation with capability argument 'struct foo' and underaligned destination (aligned ↩
    to 2 bytes) may be inefficient or result in CHERI tags bits being stripped [-Wcheri-inefficient]
       copy = memcpy(buffer, struct_with_cap, sizeof(*struct_with_cap));
```

Performance impact Currently, the compiler will always use `memcpy()` if the alignment is not statically known to be greater than the capability size. This ensures that tags are propagated if the buffer happens to be aligned at run time. For capability-containing types, this is inefficient compared to an inlined loop, so we emit a `-Wcheri-inefficient` warning that can be fixed by casting or using `__builtin_assume_aligned`. However, emitting calls to `memcpy()` for any copy greater than `sizeof(uintcap_t)` also has adverse performance impacts for small data structures that are guaranteed not to contain capabilities.

We have not yet addressed this problem but at least two potential solutions exist. The frontend could emit an attribute (e.g. *no-cheri-tags*) that allows the backend to inline copies that are greater than `sizeof(uintcap_t)` but are not strongly aligned. Another option would be to allow capability-size loads and stores to operate on unaligned addresses (or have the kernel emulate those accesses). Loads can always copy the raw data bits, but unaligned stores of a tagged value would have to trap as the tag bit cannot be split.

3.2.5 Pointer comparison

In pure-capability C/C++, comparisons of pointers and `uintptr_t` (which is a capability type and not an integer) not only compare the virtual addresses of the two pointers, but also their tags. For equality comparisons, two pointers are equal if their tags and virtual addresses are equal. For relational comparisons, we order by the address, but all untagged capabilities compare less than tagged capabilities. It is important to note that other capability metadata, such as bounds and permissions, are not used in pointer comparison.

This definition of equality means that 'pointers with different provenance [can] compare equal, but not be interchangeable' [148]. To address this concern, we added a `CExEq` instruction that checks all bits in the capability for identity and can be used for equality comparisons in C by passing the `-cheri-comparison=exact` compiler flag. However, enabling this mode resulted in hard-to-debug compatibility issues. For example, developers may attempt to compare the return value from `realloc()` to the previous pointer to detect whether the underlying allocation has changed. Yet, with strict equality the resulting capability will also compare unequal if the existing memory range was reused, but the bounds were changed. As the strict equality semantics are not compatible with existing code, we did not enable them by default.

Nevertheless, the more relaxed definition that ignores bounds and permissions also causes surprising run-time failures. One example that we saw in NGINX's `printf()` implementation was the use of `(void*)-1` as a sentinel value. NGINX used this sentinel to read until the first null character without having to determine the end of the buffer first since they assumed that `(void*)-1` is greater than any other pointer. For pure-capability C/C++, untagged values compare less than any valid capability, so the loop would instead terminate on the first iteration. We had to change this code to a virtual address comparison to work around this unanticipated behaviour.[11]

Another case where the inclusion of the tag bit in comparisons can cause surprising behaviour differences is when the resulting pointer is so far out-of-bounds that it becomes unrepresentable (see Section 3.4.2). For example, `p1 + offset < p2` could cause the left-hand-side to become untagged (which compares less than any tagged value), whereas the seemingly equivalent `p2 - p1 > offset` will not suffer from this problem.

The definition of equality also caused other issues which will be explained in detail later: integer-to-capability conversions (see Section 3.5.2) and the use of using low pointer bits to store additional metadata (see Section 3.6.1.2).

We find this definition is not only surprising, but also insufficient. Two pointers that compare equal are not necessarily equi-dereferenceable. While they will both point to the same address and have the tag bit set, they might have differing bounds and permissions. Therefore, I changed the definitions for equality and relational operators to compare

[11] This comparison compares two distinct objects and is therefore undefined behaviour according to the C standard [112, §6.5.8.5]. However, we believe that this is not the only case where the assumption that `(void*)-1` compares greater than any other pointer is being made.

only the address and recommend the use of `CExEq` for any comparison that requires the results to be used interchangeably.[12] However, it may sometimes be useful to use a stricter model that compares the entire capability. We will retain the `-cheri-comparison=exact` compiler flag to opt into this mode.[13]

3.2.6 Pointer permissions

Another source of incompatibility comes from CHERI's ability to enforce permissions on pointers. For example, it is impossible to derive a writable pointer from a code pointer. Therefore, if we enforce a read-only program counter, we cannot access writable data by deriving from it. However, some architectures use this strategy for language-invisible, implicit pointers to global variables (see Section 4.2 for more details and a solution).

The compiler could also use pointer permissions to enforce language-level constructs such as `const` by removing write permissions. However, we realize that this would cause too many compatibility issues, due to certain core APIs such as `strchr` converting `const` pointers to mutable ones [41]. Moreover, the monotonicity guarantees provided by CHERI would break existing constructs such as `const_cast` unless an authorizing capability was provided to re-derive a mutable pointer.

For non-CHERI-aware code, the pointer permissions will generally match the MMU-enforced page table permissions, and so far we have not encountered any compatibility issues caused by pointer permissions in real-world code. However, capability permissions have been very useful in tracking down bugs in the compiler, linker and the C runtime environment (see for example Appendix C.2.2).

3.3 CHERI capability registers

When adding CHERI to a processor, there are two options for implementing capability registers. We can either use a *merged register file* (extending integer registers to capability-width) or a *split register file* (use distinct registers for capabilities, similar to floating-point registers). For CHERI-MIPS we chose the latter approach, and this choice has implications for software compatibility and potentially performance (see also Section 6.2.7).

3.3.1 Function prototypes and calling conventions

The CHERI-MIPS pure-capability function calling convention uses different registers for passing pointer and integer arguments. This can lead to compatibility problems

[12] We also considered making the tag bit the least-significant bit in the comparison instead of the most-significant one. While this would improve compatibility by making e.g. `(void*)-1` compare greater than any valid pointer, it still does not provide a total order for capabilities. Additionally, it is unclear if any case exists where including only the tag bit, but not other metadata is useful.

[13] This flag currently does not change relational operators. Upper and lower bounds can be compared, but it is unclear what ordering should be given to permission bits.

if function prototypes are not declared correctly. These problems are not limited to CHERI-MIPS, it can also cause issues on other architectures where the calling convention requires integers and floating-point values to be passed in different registers. However, programmers are generally aware of calling convention issues with floating-point values, whereas some (especially older) C programming styles assume integers and pointers can be used interchangeably.

3.3.1.1 Unprototyped (*K&R*) functions

When calling functions without prototypes the compiler will use the appropriate registers for the argument data type at the call site. This can cause issues if the passed arguments do not strictly match the definition. For example, using zero as a `NULL` pointer constant (without casting it to a pointer type first) will cause the value to be passed in the next integer argument register, thus causing all following arguments to be in incorrect registers.

We saw many run-time crashes due to mismatched arguments in the FreeBSD version of `less`, which did not use any functions prototypes. To catch these issues at compile time, the compiler will now warn when a function without a declared prototype is called.[14] This warning (`-Wmips-cheri-prototypes`) is less strict than `-Wstrict-prototypes`[15] and can be used to convert only the *K&R* functions that may cause problems instead of having to change all occurrences. We therefore enable it by default when compiling pure-capability code. While this problem should not be an issue for C code written in the last 20 years, many core operating-system components can be significantly older. Additionally, `-Wstrict-prototypes` is not enabled by default and functions with an empty argument list can take any number of arguments in C — whereas C++ assumes no arguments if an empty argument list is specified. Therefore, it is common to see unprototyped functions even in modern code. Forcing `-Werror=strict-prototypes` for pure-capability C would have required many unnecessary — at least in terms of functionality — changes.

3.3.1.2 Variadic argument handling

The pure-capability calling convention passes all variadic arguments via the stack and accesses them via an appropriately bounded capability. This provides memory-protection benefits but means that variadic functions must be declared and called via a correct prototype. Some C code assumes that the calling convention of variadic and non-variadic functions is sufficiently similar that they may be used interchangeably and relies on this assumption to implement optional arguments in C. Historically, this included the FreeBSD kernel's implementation of `open()`, `fcntl()`, and `syscall()`. For pure-capability C/C++ we must instead read arguments using `va_arg` before calling an appropriately prototyped function.

[14] If the *K&R* function is defined within the same file, the compiler can determine the correct calling convention and will not emit a warning.

[15] Fixing all instances of `-Wstrict-prototypes` may require many changes to a legacy codebase.

The use of bounded capabilities for on-stack arguments has also allowed us to find bugs in various `printf/scanf` implementations and call-sites (see Section 6.1.4 or [53]) where incorrect data types were being read using `va_arg` for the `%p` (pointer) modifier.

Use of variadic functions for optional arguments Some C functions such as `open()` or `semctl()` use variadic function signatures to indicate an optional argument. This works on architectures where the non-variadic and variadic calling conventions are the same for a given number of arguments but may cause problems for pure-capability CHERI-MIPS.

In the case of `open()`, the pure-capability calling convention allowed us to find various cases where `open()` was called with the flag `O_CREAT` but without the `mode` argument which is mandatory in this case. On a conventional architecture, this results in a file being created with the mode set to whatever happens to be in the next argument registers. However, for pure-capability C this results in a run-time trap when trying to read non-existing arguments from the stack as the register for on-stack arguments will be `NULL`.

The `semctl()` case is particularly interesting as the optional argument must be of type `union semun()`, yet most code passes an integer constant instead (for example `semctl(id, num, SETVAL, 2)`). This is also incorrect on other architectures, but because the first few arguments for variadic functions are passed in registers, it is not noticeable whether a pointer or integer type is being passed. For CHERI-MIPS all arguments are passed on the stack so they have accurate size information. Therefore, passing an integer crashes at run time when trying to read a capability-size `union semun` from the bounded variadic argument capability (which only spans a 4-byte `int`).

Casts between function-pointer types Casts between function pointers with different prototypes can also cause problems which we discovered for example in the external data representation (XDR) code in libc. The XDR code liberally casts between variadic and non-variadic functions and invokes them assuming that the calling conventions are compatible. As this causes difficult to debug run-time failures, I added a new warning to Clang for casts between variadic and non-variadic function pointer types. However, it turns out that in many cases unprototyped function pointers are used to work around deficiencies in the C programming language. For example, PostgreSQL (see Section 6.1.3.4) uses unprototyped function pointers to allow polymorphic callbacks. This warning is off-by-default and can be enabled by passing `-Wmips-cheri-prototypes-strict`.

3.3.2 Recommended approach

Based on our experience with the split register file in MIPS, we believe that a merged register file can avoid many compatibility problems (mostly related to calling convention). Additionally, Section 3.8.1 presents new instructions that I added to the CHERI-MIPS ISA — some of which would not have been required if we had a merged register file — as

there would be no need to move values between capability and integer registers. The split register file also greatly complicated Khilan Gudka's efforts to port WebKit. In the current version we implement JavaScript values as capabilities, which meant adjusting many cases in the interpreter to correctly move between register files.

Which of the two approaches is better for performance is highly workload-dependent: a split register file can be beneficial if register pressure is the limiting factor, but a merged register file will improve code density and reduce the amount of data to be stored for context switches. This is a question that will be answered when we have CHERI-RISC-V prototypes with a merged register file and a mature CHERI-RISC-V compiler.

3.4 CHERI capability precision

The original version of CHERI used 256-bit capabilities. This representation allows byte granular bounds for every allocation size and provides many available permission bits. However, increasing the size of pointers by a factor of four results in large increases in cache pressure. Therefore, the latest versions of CHERI use a 128-bit capability format by exploiting redundancy between the bounds and the pointer value [118, 246, 254]. This compression scheme still allows byte-granular bounds for small allocations, has enough space for all currently used permissions and supports sealing capabilities with an object type. One constraint, however, is that larger allocations must have a minimum alignment to be precisely representable.

3.4.1 Choosing the precision for CHERI-128

Originally, CHERI-128 used 23 bits of precision as that was the smallest size allowing CheriBSD to boot without imprecise capabilities. However, this required 45 ($2 * 23 - 1$) of the 64 available metadata bits for bounds information. As this leaves little room for future ISA changes, we analysed the sizes of objects and functions both statically and dynamically to determine a sensible mantissa size for the 128-bit CHERI encoding [254].

As part of this precision study, I instrumented the compiler to collect bounds information for globals, stack allocations, sub-objects and functions. Additionally, I analysed the Flattened Device Tree (FDT) files included with the Linux and FreeBSD kernels to determine the sizes of memory-mapped I/O regions. The analysis showed that almost all capabilities can be represented exactly if we use 12–16 bits of mantissa. For further information on this study see the full report [255]. Based on this information we chose 14 bits of precision (a total of 27 bits for bounds information) as this size allows all sub-page-size allocations to be bounded precisely. Additionally, the previous capability format had to resort to different representations for sealed and unsealed capabilities, but by reducing the bounds precision we can include the object type field for all capabilities. In the currently chosen format, up to 4095-byte objects can be represented exactly and

any larger object must be aligned to at least 8 bytes. This alignment requirement scales linearly with the size of the object.

3.4.2 Out-of-bounds pointers

Whilst the C standard only permits pointers to be within bounds or point to one element past the end, in practice code does not adhere to these rules. According to the C standard, relational comparison for pointers is only defined if both pointers point to the same object or array or are one-past-the-end pointers [112, §6.5.8.5]. Similarly, (in-)equality comparisons are also undefined for pointers that are not in bounds or one-past-the-end [112, §6.5.9.6].

We believe that it should be possible to implement all C programs using CHERI capabilities and not just the subset adhering to an idealized standard. Therefore, it is possible for CHERI capabilities to hold out-of-bounds values.[16] In the 256-bit representation this is straightforward as base, length and offset are 64-bit values and therefore any out-of-bounds pointer can be created. This is slightly more complex using the 128-bit format [254], but unlike other compressed fat-pointer schemes (such as Low-Fat [132]) there is explicit support for out-of-bounds pointer values. When a pointer goes out of bounds, the redundancy between address and base is reduced, and at some point the bounds can no longer be represented. The exact number of bytes that a pointer can be out-of-bounds by depends on the number of bits used for precision and whether it points before or after the object. In the current encoding scheme, pointers up to 2KiB (or 1/4 of the object size if that is greater) after the end and 1KiB (or 1/8 of the object size if that is greater) before the start of the bounds can be represented. When a capability is no longer representable, the tag bit is cleared, and no further accesses are possible using this invalidated pointer.

Support for out-of-bounds pointers adds complexity to the encoding scheme and reduces the number of bits in the limited 128-bit capability encoding that could be used for other purposes.[17] As part of exploring pure-capability C, we therefore verify the hypothesis that supporting transiently out-of-bounds pointers is necessary to support running C/C++ based operating systems and applications. Prior work in 2015 used an instrumented compiler to show that many programs include out-of-bounds intermediate values [41]. Almost four years later, we are able to run code in a pure-capability environment and can verify that this analysis applies at run time (i.e. the code sequences found by static analysis are actually executed).

To determine the number of out-of-bounds pointers that are used by existing C applications, I modified QEMU to gather statistics when creating out-of-bounds pointers. This is possible without false positives on CHERI since capability manipulation must

[16] These out-of-bounds values can only be dereferenced if they are brought back into bounds using pointer arithmetic.

[17] Naturally, this also applies to the 64-bit encoding with a 32-bit address space.

be performed using special instructions instead of integer instructions. We collected statistics for out-of-bounds pointer creation during a full run of the FreeBSD test suite. We encountered 155214 pointers that were out-of-bounds by more than one byte. Of these, only 1205 pointed past the end. This indicates that despite construction of one-past-the-end pointers being legal, having pointers to before the object is more common in real-world code. We also found that 81% of these pointers are at most `sizeof(void*)` beyond the bounds. While this is a surprising number of out-of-bounds pointers, there were at most six unique program counters generating these per process. One typical use of an out-of-bounds pointer is an idiom in which an array pointer is incremented prior to use within a loop, leading programmers to decrement the pointer below the lower bound in the loop. This idiom can be found in the `zlib` compression library, affecting dependent software such as `gzip`, OpenSSH, and `libpng`.

While six unique program counters per program may seem like a negligible amount of code to be changed, every additional restriction imposed by pure-capability C/C++ on existing code increases the cost of adoption and therefore makes it less likely that CHERI could be deployed outside of an academic setting. Therefore, we conclude that the additional encoding bit used to represent out-of-bounds pointers is a good trade-off since it allows more source code to compile and run unmodified.

3.4.3 Compatibility concerns due to precision

The precision of capabilities has only minor implications for language-level compatibility with existing code. The only time we saw that an application was affected by precision happened in PostgreSQL (see Section 6.1.3.4), which was requesting a large, oddly sized allocation from `mmap()`. The `mmap()`/`munmap()` API is awkward for compressed capabilities since `munmap()` relies on the caller's notion of size. The `mmap()` interface provides no way to return the length actually mapped; so if mappings were automatically rounded for precise representability, and users attempted to unmap with a non-rounded size, the last few pages might remain mapped. To avoid this problem, `mmap()` returns an error if the allocation size is not precisely representable, and the call-site must be changed to request an appropriately aligned and sized mapping [53].[18]

When the compressed capability format was introduced, we added a `CSetBoundsExact` instruction that faults whenever the bounds cannot be represented exactly. The existing `CSetBounds` instruction rounds down the base of the resulting capability and rounds up the length if it is not precisely representable. However, this can result in capabilities to different objects overlapping if they are large enough. In the case of CHERI-aware code that explicitly sets bounds (e.g. memory allocators, RTLD or the `mmap()` kernel implementation), using `CSetBounds` on allocations may result in rounding, thus potentially granting access

[18]This problem has since been fixed by modifying the kernel, but it still existed at the time this dissertation was written.

to a few bytes of adjacent allocations. While these bounds are a substantial improvement over non-CHERI systems, it would be ideal to not allow overlapping capabilities. We have adjusted the CheriBSD allocators (and sub-allocators) to align and pad allocations to prevent any accesses to unrelated allocations. However, programs with custom memory (sub-)allocators will return poorly bounded capabilities (using bounds from the underlying allocator, e.g. the entire `mmap()` range). This is not a compatibility concern; however, it does mean these applications or libraries have weaker memory protection than other pure-capability code using the system allocator. For this reason — and to fix compatibility issues caused by poor alignment (see Section 3.2.3) — it would be very useful future work if the compiler (or a static analyser) could locate (sub-)allocators.[19] Currently, this is a manual process, and we may not have found all sub-allocators in the codebases that we have ported. However, the sub-object protection that I introduce in Chapter 5 automatically narrows bounds in many more cases and should therefore provide additional protection even if the memory (sub-)allocator does not set bounds.

3.4.4 ISA extensions related to precision

instruction that faults whenever the bounds cannot be represented exactly. The existing `CSetBounds` instruction will round down the base of the resulting capability and round up the length if it is not precisely representable. However, this can result in capabilities to different objects overlapping if they are large enough. Memory allocators (and the `mmap()` kernel implementation) must take care to round up the requested allocation size and alignment so that the padding required for capability representability does not overlap with another object. Correctly computing the rounded size and minimum alignment for a given allocation is non-trivial and requires many instructions to compute.[20] To make it easier for allocators to round allocations to representable sizes, I introduced two new instructions to the CHERI ISA and added matching compiler built-ins. The first is `CRoundRepresentableLength` (`CRRL`)[21], which takes a length and returns the size that it would be rounded to when used in `CSetBounds` (assuming appropriate alignment of the base). The second is `CRepresentableAlignmentMask` (`CRAM`), which returns a bitmask that can be used to align an address downwards such that it is sufficiently aligned to create a precisely bounded capability. For full semantics see the CHERI architecture specification [246, §D.2 and §D.20].

Performance gains from these new instructions To verify the effectiveness of these new instructions I wrote a micro-benchmark [187] that allocates 4096 objects using

[19] A tool for dynamic detection in x86 binaries exists [38], but we are not aware of any compile-time tool.

[20] This is especially noticeable on the MIPS version that CHERI is based on since it lacks count-leading-zeroes and count-trailing-zeroes instructions.

[21] Formerly known as CRoundArchitecturalPrecision but renamed by popular demand.

`malloc()`, then `free()`'s every third allocation and then calls `realloc()` with a different size on the remaining allocations. `malloc()` and `realloc()` have to round up sizes to prevent capability bounds from granting access to other objects due to imprecision. Prior to the introduction of these new instructions we had to use a complex instruction sequence to round allocations to a precisely representable size. The performance improvements from are most noticeable in `malloc()`. For this benchmark, `CRoundRepresentableLength` reduced the instructions executed when calling `malloc()` by 5.95%. For `realloc()` this effect is less pronounced (as it does not affect the `free()` part of `realloc()`), yet still reduces the instructions executed by 2.03%. Additionally, `CRRL` provides more accurate rounding of the required length. The previous hand-written code was an over-approximation, which means we no longer unnecessarily add padding bytes.

3.4.5 Compiler changes due to reduced precision

The precision constraints of CHERI capabilities should rarely be visible to programmers. However, the compiler must take them into account for the layout of variables if we want to guarantee non-overlapping capabilities. To achieve this goal, we modified LLVM to increase alignment and insert padding for all global variables and stack allocations when required by the *CHERI Concentrate* encoding [118, 254]. These changes were small since we compute the required alignment and padding using the existing CHERI Concentrate library [184] that I originally wrote for QEMU.[22]

Sometimes programmers allocate dynamically sized arrays on the stack (e.g. by using variable-size arrays or calling the `alloca()` function). In this case we also need to align down the stack to ensure that the capability bounding this dynamic allocation has precise bounds that do not overlap with any other object. Prior to the introduction of the `CRepresentableAlignmentMask` instruction, we simply aligned down the stack by 128 times the size of the array type to ensure the bounds are representable. However, this results in wasted stack space: allocations smaller than 4096 bytes do not need any additional alignment. By modifying the compiler to align down the stack with `CRepresentableAlignmentMask` instead of the hard-coded limit, we no longer waste stack space for dynamic allocations. This is especially noticeable with small allocations that do not require any additional alignment, but even for larger allocations we would only need the previous minimum alignment of 128 if the allocation is larger than 64KiB.

3.5 Conversions between capabilities and integers

Converting between integers and capabilities can result in surprising run-time behaviour. This section will highlight some concerns when converting between integer constants

[22]Besides being used in CHERI-MIPS QEMU and LLVM, this library has since also proven useful for the GDB debugger (to decode capability values) and the CHERI-RISC-V simulator.

and pointers, and Section 3.6 will focus on conversions between `uintptr_t` (which is a capability in pure-capability C/C++) and plain integers.

3.5.1 Conversions in hybrid CHERI C

Before looking at pure-capability CHERI C, I will first present an important compatibility issue discovered in the hybrid compilation model. While this may seem unrelated to pure-capability CHERI, this issue shares similarities with a compatibility problem in pure-capability mode. Both are caused by data-type conversions implicitly being relative to some other value and therefore yielding surprising results.

Initially, the compiler allowed implicit conversions between integer pointers and capability pointers (those annotated with `__capability`). However, performing such a conversion and casting back to the initial type does not always return the initial value. To convert between capabilities and pointers, the compiler uses instructions that perform the conversion relative to the default data capability `$ddc` [41]. This can result in an incorrect capability or `NULL` if the value to be converted originates from a domain with a different `$ddc`. Therefore, I added a new compiler warning that triggers when converting between capability and integer pointers. To explicitly request the conversions, I extended the language with two new casts: `(__cheri_fromcap T*)` converts from capability to integer and `(__cheri_tocap T* __capability)` from integer to capability. This warning has been useful in the CheriBSD kernel, the largest hybrid C program that we currently compile. In the future the kernel `$ddc` will no longer overlap with the `$ddc` of userspace programs, so any `$ddc`-relative conversion of user pointers will result in an incorrect address. In almost all cases the desired value is actually the absolute address, so the new warning avoids surprising run-time errors.

Later, we discovered another problem with the CHERI instruction used to implement capability-to-integer conversions (`CToPtr`). The header <sys/signal.h> declares constants to be used as signal handlers: e.g. `SIG_DFL` for the default behaviour and `SIG_IGN` to ignore signals. The definition of `SIG_IGN` was `(void*)(__uintcap_t)1`. Surprisingly, this did not result in a pointer with value 1 but a `NULL` pointer.[23] This is caused by the compiler seeing a conversion from a capability type (`__uintcap_t`) and therefore using `CToPtr` to convert to an integer. However, `CToPtr` was designed for interaction between capability sandboxes and non-capability-aware code so yields NULL for untagged capabilities to avoid security issues [245]. As this is extremely difficult to debug due to the valid-looking result from `CToPtr`, I modified the compiler to emit a warning whenever it generates a `CToPtr` instruction without an explicit `__cheri_fromcap` cast.

[23] To make things even more surprising, assigning `SIG_IGN` to a global variable would yield the expected value and fail only when used inside functions (or when compiler optimizations elided the global variable).

3.5.2 Integer-to-pointer casts in pure-capability C

In C and C++ programmers sometimes cast an integer literal to a pointer. However, it is not clear whether such expressions should result in valid pointers or not. We have seen the following three cases:

- Kernel code sometimes casts from integer constants to `memory_mapped_device*`. The programmer expects this to be a valid pointer that can be dereferenced. In this case we would need to create a `$ddc`-derived capability.
- Constants are also sometimes used to trigger a trap with a known faulting address. This does not need to be a valid pointer, so a `NULL`-derived value can be used.
- Finally, these expressions are also be used as special marker values. Locking code sometimes uses expresssions such as `void* locked = (void*)0x1` in fast paths. For the slow path, the value is replaced by a valid pointer that can be dereferenced. For these constants, the value should not be derived from `$ddc` but instead be an untagged `NULL`-derived capability.

Originally, the compiler created `$ddc`-relative capabilities for integer-to-pointer casts inside functions and generated an untagged value for global variables. This worked in most cases but caused surprising issues. We discovered that this behaviour broke the QtBase `QReadWriteLock` when compiled with optimizations (but not when optimizations were disabled). `QReadWriteLock` uses the following variable declarations:

```
namespace {
  enum { StateMask = 0x3, StateLockedForRead = 0x1, StateLockedForWrite = 0x2 };
  const auto dummyLockedForRead = (QReadWriteLockPrivate *)(uintptr_t)StateLockedForRead;
  const auto dummyLockedForWrite = (QReadWriteLockPrivate *)(uintptr_t)StateLockedForWrite;
}
```

As the `dummyLockedForRead` and `dummyLockedForWrite` variables are in an anonymous namespace, the compiler does not need to allocate storage and can inline them at the use-site. When inlined, the compiler emitted `CFromPtr $ddc, 0x2` and therefore created a valid capability instead of the untagged constant.[24] Later, this value was compared to `StateLockedForWrite` for which the compiler generates an untagged value. As the tag bit of the two values did not match, the comparison failed (see Section 3.2.5) and the code assumed it had a valid pointer instead of the special marker constant. It then dereferenced that pointer and unsurprisingly crashes when accessing memory at address `0x2`.

I fixed this problem by making the compiler consistently use `NULL`-derived capabilities for all integer constants cast to pointers or `uintptr_t`. This breaks the case where kernel programmers want to create a pointer to a memory-mapped device, but the compiler will warn that the result of an integer cast to a pointer is not dereferenceable. Additionally, it is not necessarily true that `$ddc` is the correct capability from which to derive the

[24] In the current version of CheriBSD this would be a NULL-derived value as we run pure-capability programs with a `NULL $ddc`. However, this problem was discovered when we were still using the legacy CHERI linkage model (see Section 4.2.2) and `$ddc` held a full address-space capability.

access, so the compiler warning should tell the programmer to manually derive from a valid capability spanning that memory region instead.

3.6 Offset and address interpretation of capabilities

The original design of CHERI capabilities only included a *base* and a *length* field and any occurrence of pointer arithmetic would reduce the accessible region [256, 257]. Later, the design and representation was changed to add a capability *offset*, which allows CHERI capabilities to behave more like *fat pointers*. The added offset improved support for pointer arithmetic and common C idioms such as storing data in low bits of aligned pointers [41].

The C language model for capabilities was focused on the idea of not leaking virtual addresses in order to make copying garbage collection possible. This resulted in capability *offsets* being used wherever possible instead of exposing the address. For example, casting a `uintptr_t` to a `long` yielded the offset from the base of the underlying object. This value remains stable even if the underlying object has been relocated to a new location by a garbage collector. This choice is also visible in the ISA: most instruction names refer to offsets and not addresses, e.g. `CIncOffset` rather than `CIncAddr`.

The use of offsets can be beneficial for a garbage-collected C implementation, but in most cases C programmers expect addresses and not offsets relative to a base. In this section I highlight some problems with *offset* interpretation of capability and show that it is not sufficient for (near) source-level C compatibility. I first showcase the approaches I used to deal with the compatibility problems and then present the solution, the *address* interpretation of capability.

3.6.1 Bitwise operations on capabilities

The most common source of incompatibilities was bitwise arithmetic on `uintptr_t`. In most cases bitwise operations — such as those used to store or clear flags in the lower bits of pointers to well-aligned allocations — will result in the expected `uintptr_t` value being created. However, in some corner cases the result may be unexpected. As this can result in hard-to-debug failure modes, I added a `-Wcheri-bitwise-operations` compiler warning that triggers when using bitwise operations on capability types such as `uintptr_t`. To fix this warning we now provide higher-level abstractions for bitwise operations on `uintptr_t`, which usually fall into one of three categories: alignment, low pointer bits and hashing.

3.6.1.1 Changing and checking pointer alignment

Uses of bitwise-`AND` to determine alignment may result in an incorrect value in pure-capability C/C++ using an offset interpretation of capabilities. For example, checking whether a pointer is aligned often uses code like this: `(vaddr_t)ptr & 15 == 0`. The cast to `vaddr_t` always returns zero for capabilities pointing to the beginning of an

allocation (i.e. address equal to base) and therefore the check succeeds even if the virtual address is not aligned. This problem was first noticed in 2016 [148], but a solution was only found much later.

Similarly, the commonly used bitwise arithmetic to align pointers up and down will adjust only the offset, thus resulting in an incorrectly aligned pointer if the base is not sufficiently aligned. We saw that this broke e.g. the memory allocators in NGINX and PostgreSQL. To work around this issue, I introduced new compiler built-ins that can be used instead of hand-written bitwise operations:

`_Bool __builtin_is_aligned(T ptr, size_t alignment)` returns `true` if `ptr` is aligned to at least `alignment` bytes.

`T __builtin_align_down(T ptr, size_t alignment)` returns `ptr` rounded down to the next multiple of `alignment`.

`T __builtin_align_up(T ptr, size_t alignment)` returns `ptr` rounded up to the next multiple of `alignment`.

One advantage of these built-ins compared to `uintptr_t` arithmetic is that they preserve the argument type and therefore remove the need for intermediate casts to `uintptr_t`, which can result in types accidentally being changed or qualifiers such as `const` being dropped. I have submitted these changes to LLVM as these built-ins are useful not only for CHERI, and the upcoming version 10 release will include them.

3.6.1.2 Storing additional data in pointers

In many cases the minimum alignment of pointer values is known and therefore programmers assume that the low bits (which will always be zero) can be used to store additional data.[25]

Checking vs. clearing of low pointer bits Clearing and setting low bits in pointers to store additional metadata is a common technique in C and C++ as most pointers will always contain zeroes in the lower bits. However, the offset interpretation of capabilities (and the inclusion of tag bits in comparisons, see Section 3.2.5) may result in surprising run-time behaviour differences. We discovered that the use of low pointer bits caused `QMutexLocker` (used for the mutex class in the Qt framework) to deadlock. `QMutexLocker` takes a pointer to a mutex and is supposed to unlock it in the destructor but in pure-capability C++ it was not being unlocked correctly.

```
// first & used to get low bits:
if ((mtx & (uintptr_t)1u) == 1u) {
  // second one to clear the bits:
  mtx &= ~(uintptr_t)1u;
  do_unlock((QMutex*)mtx);
}
```

Listing 3.2: Bitwise-`AND` resulting in deadlock inside `QMutexLocker`.

[25] CHERI actually provides many more usable bits than a conventional architecture: In the current implementation of 128-bit CHERI, any bit between the least significant and the 9th least significant bit may be toggled without causing the tag to be cleared in pointers that point to the beginning of an allocation. If the pointer is strongly aligned, it may be possible to store even more additional bits.

The `QMutexLocker` constructor locks the underlying mutex and sets the lowest bit to indicate that it is locked using `(uintptr_t)mutex |= 1`. In the destructor, it checks whether the low bit is set and if so unlocks the mutex. A simplified version of the code in the destructor can be seen in Listing 3.2. As can be seen in the listing, there are two different uses of the bitwise-`AND` operator. In the first case, it is used to retrieve the low bits of the pointer. However, the pure-capability C/C++ offset-centric semantics cause this operation to return `mtx` with the offset set to `getoffset(mtx)&1`. This yields a valid pointer, which is clearly distinct from a `NULL`-derived `(uintptr_t)1`. Therefore, the conditional statement was never entered, and the unlock function was never invoked.

This problem could be fixed by deriving bitwise-`AND` results from `NULL`. However, this would break the second statement in this code example. After checking whether the low bit is set, `QMutexLocker` clears the low bit and passes the original `QMutex` pointer to the unlock function. In this case the result must inherit provenance from the left-hand-side.[26]

This ambiguity means that the compiler cannot correctly select which operand of a bitwise-`AND` expression should be used as the provenance source (see also Section 3.10.2). When clearing bits to retrieve the pointer, we should inherit the provenance of the pointer, but when retrieving the low bits, the result should be a `NULL`-derived capability.

Compiler warnings and explicit macros I initially attempted to solve this ambiguity by emitting compiler warnings for every use of bitwise-`AND` on capabilities. I also added explicit macros for storing additional data in low pointer bits to the compiler-provided header `cheri.h`.[27] In addition to fixing the compiler warning, the use of these macros can improve readability of code manipulating low pointer bits.

`uintptr_t cheri_set_low_ptr_bits(uintptr_t ptr, vaddr_t bits)` This function returns `ptr` bitwise-ORed with `bits`. To retain compatibility with non-CHERI architectures, `bits` should be less than the known alignment of `ptr`.

`vaddr_t cheri_get_low_ptr_bits(uintptr_t ptr, vaddr_t mask)` This function returns the low bits of `ptr` in the same way as `ptr & mask`. It should be used instead of the raw bitwise operation since it can never return an unexpectedly tagged value. `mask` should be a bitwise-AND mask less than `_Alignof(ptr) - 1`.

`uintptr_t cheri_clear_low_ptr_bits(uintptr_t ptr, vaddr_t mask)` This function clears the low bits of `ptr` in the same way as `ptr & ~mask`. It returns a new `uintptr_t` value that can be used for memory accesses when cast to a pointer. `mask` should be a bitwise-AND mask less than `_Alignof(ptr) - 1`.

Data-dependent provenance While the explicit use of macros fixes the provenance source ambiguity with bitwise-`AND` operations, it requires many code changes, which we

[26] This problem is not solely caused by *offset* interpretation of capabilities, but also by the inclusion of tag bits in pointer equality (see Section 3.2.5).

[27] The similar pattern of storing data in unused high pointer bits is not supported (see Section 3.9.5).

would like to avoid. As an alternative work-around, I introduced an experimental flag `-cheri-data-dependent-provenance`. When enabled, the compiler assumes that all bitwise-`AND` operations with a small integer constant are checking the low pointer bits rather than attempting to clear bits to obtain a valid pointer. The compiler inserts checks whether bitwise-`AND` operands are less than 4096 (since pointers to the first page are unlikely to be valid) and if so, it derives the result from `NULL` instead of the left-hand-side operand. We have used this flag for various projects, and it works as expected, thus removing the need for many source-code changes. However, it results in a surprising and inconsistent model for provenance and therefore we will not enable it by default.[28]

Additionally, we have since introduced an address interpretation of capabilities (see Section 3.6.4) and discovered a better solution. The underlying issue is caused by pointer comparisons including the tag bit. As this has caused many incompatibilities, we adjusted the comparison semantics to no longer include the tag (see Section 3.2.5).[29]

3.6.1.3 Computing hash values

Another case where arithmetic on `uintptr_t` caused problems was modulo or shift operations. This usually indicates that the pointer is being used as the input to a hash function or similar computations (modulo may also be used for alignment checks). Using the offset is almost never correct here, as a conventional C/C++ implementation would perform the operations on the address.

In these cases, the programmer should not be using `uintptr_t` but instead cast the pointer to `vaddr_t` and perform the arithmetic on this type instead. This has the advantage that it will be more efficient than `uintptr_t` arithmetic on a split register file architecture such as CHERI-MIPS. I therefore modified the compiler to emit warnings for shifts or modulo operations on `uintptr_t` that suggest using `vaddr_t` instead.

3.6.2 Implicit conversions between `uintptr_t` and integers

As with explicit casts, implicit conversions from `uintptr_t` to any integer type return the offset. These implicit conversions are often not visible in the source code and can result in surprising failure modes. This property broke e.g. the code in RTLD responsible for relocating the GOT. RTLD was adding `(uintptr_t)relocbase` to all GOT entries to obtain the actual run-time address. When we started using tightly bounded capabilities for the mappings instead of a full-address-space capability, the offset was zero instead of the desired address, so RTLD filled the GOT with incorrect values, resulting in run-time

[28] I instead added an alternative solution (see Section 3.10.2) shortly after the initial submission of this dissertation.

[29] Interestingly, using addresses instead of offsets in combination with CHERI-128 compression (see Section 3.4) causes the checking of low pointer bits to behave 'as expected' in most cases. When performing a bitwise-`AND` with a small value, the pointer address will almost always (except in cases where the pointer value is near the NULL page) be so far out of bounds that the tag bit is cleared. This ensures that the comparison to an untagged integer constant evaluates to true.

crashes.[30] We also found various cases in the CheriBSD kernel where implicit offset-yielding conversions from `uintptr_t` to the address type `vm_offset_t` resulted in crashes and in libunwind these implicit conversions caused C++ exceptions to not function correctly.

Attempted solution Since this was a common error, I added a new attribute to Clang that indicates that a conversion to this type should yield the address instead, `__attribute__((memory_address))`. I also added a new diagnostic to Clang to detect casts from capabilities (i.e. pointers, `intptr_t` and `uintptr_t`) to plain integers. This new warning will trigger whenever a capability is cast to an integer type that has not been annotated with `__attribute__((memory_address))`. I started adding this annotation to many types that are used for addresses such as `vaddr_t`, `Elf_Addr` and many more. However, it turns out that programmers often use types such as `size_t` when casting pointers to virtual addresses and `size_t` cannot be annotated as being an address type. This new warning was extremely noisy and required many source-level changes, so we decided to keep it disabled by default.

3.6.3 Other compatibility concerns

In addition to the two major issues listed above, we also found some less common problems.

Switch statements One of the first incompatibilities that we discovered was related to `switch` statements operating on `uintptr_t`. We found a case in the FreeBSD locale code where a `switch` statement was used to handle magic pointer constants. The `switch` had special cases for `uintptr_t` values of zero and `-1` and otherwise (in the `default` case) returned the argument unmodified. However, all valid locale pointers had an offset of zero and therefore (incorrectly) did not take the `default` case.

To work around this, we changed the compiler to use the address when performing a `switch` on `uintptr_t` but only much later considered the implications of offsets on C compatibility. This behaviour is very inconsistent, as all other uses of `uintptr_t` return the capability offset.[31] At the time this was considered a special case, so we made an exception for switch statements where the offset interpretation (almost) never makes sense.

Inconsistent cast expressions Another source of confusion in the offset model is that casts from pointers and `uintptr_t` to integers are inconsistent. For a pointer `void* foo`, the expression `(vaddr_t)foo` will retrieve the virtual address of `foo`, whereas `(vaddr_t)(uintptr_t)foo` will result in the capability offset being read. This non-intuitive behaviour has caused many issues in the past and required adding the new CHERI cast modifiers `__cheri_offset` and `__cheri_addr` that allow explicit annotation of

[30] We no longer use the MIPS GOT for pure-capability binaries, but it was required in our legacy ABI (see Section 4.2.2).

[31] We are considering removing this inconsistency in the future (see Section 3.10.1).

which behaviour is desired.[32] Using `(__cheri_addr vaddr_t)var` retrieves the capability address from a `uintptr_t` and `(__cheri_offset vaddr_t)var` returns the offset.

Sub-object bounds Another case that is not compatible with offset interpretation of capabilities is sub-object bounds (see Chapter 5). When using sub-object bounds, many more pointers will have a base equal to the current address and therefore returning the offset makes little sense. We are considering adding a compiler warning (or error) when attempting to use sub-object bounds with offset interpretation of capabilities.

3.6.4 Introducing an *address* interpretation of capabilities

After introducing the many different work-arounds listed earlier, we realized that in fact many changes we had to make to source code were due to the original design choice of using capability offsets. As one of the major goals for pure-capability C/C++ is almost complete source-level compatibility, we had to find a solution to minimize changes. I therefore modified the compiler to add an optional compilation mode (`-cheri-uintcap=addr`) that would return the virtual address instead of the offset when converting from `uintptr_t` to another integer type.

When looking at code that we have previously ported to run in pure-capability mode with offset interpretation, we see that, depending on the project, around 10–50% of all changes are no longer required (see Section 6.1). Another observation supporting the default use of address interpretation is that `libc.so` contains 3.5 times more get-address and twice as many set-address instructions than the offset equivalents — even though it is compiled using the offset interpretation of capabilities.[33]

We have therefore switched the default capability interpretation to use addresses instead of offsets. The old *offset* mode can still be enabled by using `-cheri-uintcap=offset` as this may be interesting for experiments with copying garbage collection.

3.7 Adding C++ support

Once we had a pure-capability C environment running, we (David Chisnall, Khilan Gudka and I) started working on getting C++ into a well-supported state. While C++ support was in fact a significant amount of effort, the majority of these changes are internal compiler fixes and changes to runtime libraries and are therefore not of particular interest in the context of this dissertation. Due to space constraints I therefore only provide a brief subset of the changes that had to be made before we were able to run a pure-capability C++ 'Hello, World' program.

[32] We also support a `__cheri_fromcap` and `__cheri_tocap` cast that allows conversion between capability and integer pointers in the hybrid compilation mode.

[33] In address mode these ratios change to 10.5 and 6 respectively. Almost half of the offset instructions remain when compiling in address mode which indicates that these are explicit assembly/built-in uses.

- C++ references had to be implemented as capabilities.
- Pointers-to-members had to be changed to use capabilities.
- Similarly, the *vtable* pointers used for virtual functions now contain capabilities.
- Using capability types (e.g. `uintptr_t`) as template arguments previously crashed and now produces an error if the value is greater than 2^{64}.
- We added support for C++11 strongly typed enums with underlying type `uintptr_t`.
- We had to add support for `uintcap_t` integer promotion in overloaded binary operators. Additionally, we fixed the lookup of overloaded operators to handle implicit promotion from integers to `uintcap_t`.
- Some library functions such as `std::hash` require capability (`uintcap_t`) overloads.
- C++ run-time type information (RTTI) and exceptions had to be adjusted to use capabilities instead of integer pointers.
- For run-time C++ exception support, we had to port libunwind to pure-capability CHERI, teach it about new registers and fix the handling of unwind information.
- The C++ ABI library (libcxxrt on FreeBSD) also needed some changes in the DWARF parsing code.
- We also added support for sub-object bounds for C++ references (see Section 5.3).

We also had to make many changes to support the hybrid compilation model.

- C++ name mangling had to be updated to include the `__capability` qualifier. Pointers implemented as capabilities are now mangled with a `U3cap` suffix.[34]
- Function overloads need to take into account the `__capability` qualifier.
- Similarly, initialization and implicit conversion sequences must handle the qualifier.
- Finally, we added support to differentiate capability types (e.g. `void*__capability`) and integer pointers (`void*`) in templates.

One of the most surprising discoveries was that we could run almost the entire libc++ test suite successfully while a simple C++ *Hello, World* program was still crashing. It turned out that this is due to the obscure C++ feature of *pointers-to-members* being used by the libc++ <iostreams> implementation. Another interesting finding was that the hybrid-compilation mode that was easier to implement for C than the pure-capability mode, actually proved to be a lot more complicated in C++.[35]

Defending against COOP attacks A recent C++-specific attack is '*Counterfeit Object-Oriented Programming*' [198], a code-reuse attack exploiting the C++ virtual function

[34]This is not strictly required in pure-capability mode. However, the changed mangling prevents accidentally linking against MIPS libraries instead of pure-capability ones. The downside is that names are longer, and the library incompatibilities are now detected by LLD, so it might make sense to remove the qualifier for pure-capability binaries.

[35]This is not entirely true: If we had to start C++ support from scratch the hybrid compilation mode would indeed have been easier to implement. However, when looking purely at C++-specific changes after already having a working C compiler, many of the tricky changes were related to casts and conversions between capabilities and non-capabilities. In contrast, the changes to use capabilities instead of pointers (e.g. for *vtables*) were mostly straightforward.

call mechanism. The COOP attack relies on the creation of overlapping C++ objects with attacker-controlled *vtables*. While it is still possible for an attacker to create a 'fake' C++ object with a *vtable* by copying valid pointers, this would depend on an arbitrary read-write primitive being available. Pure-capability C++ enforces spatial safety, which will prevent this attack from succeeding.

3.8 Optimizations for pure-capability code

Over the past few years I have added many optimizations to the compiler to allow pure-capability code to run faster. For example, LLVM now knows about the CHERI capability intrinsics and can optimize sequences of intrinsics: setting the address followed by increment can be a single set-address and getting the address after `CSetBounds` returns the same value. These optimizations are particularly useful for arithmetic on `uintptr_t`, where it allows us to fold instructions similar to existing integer arithmetic optimizations.

3.8.1 ISA changes

In addition to the precision-related instructions mentioned in Section 3.4.4, I also made various changes to the ISA to allow pure-capability code to perform better. Most of these changes will be described in this section, but I also made some linkage-specific changes (such as the introduction of *sentry* capabilities and a capability load with a larger immediate range) which will be detailed in Section 4.7.

Address-manipulation instructions As we shifted further towards a compilation model in which we use *address* semantics for capabilities instead of *offsets* (see Section 3.6), we discovered that the generated code in *address* mode included many sequences of `CGetBase`, `CGetOffset` followed by an addition to obtain addresses. Similarly, we saw many cases of setting the address on a capability which also required three instructions and a temporary register. One (performance-critical) example that sets addresses is the initial capability relocation processing code in RTLD (see Section 4.4). Therefore, the latest version of the ISA now includes explicit address instructions `CGetAddr` and `CSetAddr`. [36]

Instructions for bitwise operations When looking at the generated code for WebKit, we discovered that there were many instructions that moved values from a capability register to an integer register, performed bitwise masking and sometimes moved it back to a capability register. The reason for this is that the type of JavaScript values is encoded in the topmost bits of a `double`. We also found the masking of pointers to be a common pattern in other code. For example, it is used to align up or down pointers or to store

[36] We do not require a new instruction for pointer arithmetic as we can still use `CIncOffset`. However, future versions of the ISA may further de-emphasize the offset notation and choose a more address-centric name such as `CIncAddr` or just `CAdd`/`CIncrement`.

New instructions						New/moved registers		
CGetAddr	CSetAddr	CGetAndAddr	CAndAddr	CRAM	CRRL	NULL	$ddc	TLS
23,093	22,160	23,352	1,743	9	28	146,516	3	316

Table 3.1: Count of new ISA feature usage in QtWebKit's DumpRenderTree binary.

additional data in low pointer bits. Therefore, I introduced new bitwise instructions that operate on the capability address. These new instructions are `CAndAddr` which applies a mask to the capability address (equivalent to `CGetAddr`, mask, `CSetAddr` sequence) and `CGetAndAddr` which returns the capability address with a mask applied [246, §D.20]. The latter would not be required on a merged register-file architecture, but for CHERI-MIPS noticeably improves code density. We did not add a bitwise-`OR` instruction (`COrAddr`) as this pattern was less common than bitwise-`AND`, but we may add it in the future.

Performance impact of new instructions Although the total uses of the new instructions (see Table 3.1) may seem low in a binary as large as `DumpRenderTree` (13.5 million instructions in total), the performance impact was significant. By adding compiler support for `CAndAddr`, `CGetAndAddr` and `CSetAddr`, I was able to reduce the instructions required to render a basic web page from 100.5 million to 92 million. The bitwise operations are especially useful since WebKit encodes every JavaScript object using *NaN-boxing* [93].[37]

Special capability registers I noticed that over 3% of the instructions in libc.so were used to generate a `NULL` value with `CFromPtr`. MIPS code can use register `$zero` as a constant value of zero in all instructions, but for pure-capability code there was no `NULL` register, which resulted in longer instruction sequences.

Originally, CHERI register `$c0` was used to access `$ddc`.[38] Additionally, certain special capability registers that are used for exception handling in the kernel (such as the kernel data capability `$kdc` or the exception program counter capability `$epcc`) were exposed in the general purpose register file as registers 27–31. This meant that each CHERI operation had to check whether it was accessing register 27–31 and potentially trigger a trap.

I therefore changed the ISA to move all special registers to a separate namespace.[39] This move frees registers 27–31 for use by software[40] and makes register zero (previously `$ddc`) available, which I repurposed to always hold a `NULL` capability.

The `NULL` register is very useful, for example to zero-initialize structure members without having to synthesize NULL in a register first, to implement inline `memset()`, or

[37]With a merged register file we could use integer bit-manipulation instructions for non-pointer JavaScript objects, but for the CHERI-MIPS split register file we need to move values between register files.

[38]This prevented reordering of CHERI instructions for optimization, since the compiler could not know before register allocation whether a CHERI instruction might modify `$ddc`.

[39]While moving special registers, I also added a new special capability register for thread-local storage (TLS) (see Section 4.7.3).

[40]The compiler currently reserves these for use by the operating-system kernel.

to create untagged `uintptr_t` values.[41] After this change, the number of instructions in libc.so that write `NULL` to a register dropped by 83% (from 16260 to 2723). Additionally, the compiler was previously saving and restoring `NULL` values, so in total this ISA change reduced the code size of libc.so by 4.2%. As can be seen in Table 3.1, the use of `NULL` is much more common than reading `$ddc`, so using zero to encode `NULL` rather than `$ddc` improves code density.[42] Another side effect of the special-purpose capability register change (although not the primary motivation) is that it simplified the FPGA implementation and made the QEMU emulator faster since it can omit the register access checks previously required for every instruction.

3.8.2 Optimizing bounds on stack variables

While looking at QtWebKit performance I noticed that we were generating more instructions and memory accesses for CHERI compared to MIPS. One of the main sources of this turned out to be the bounding of stack allocations. The CHERI compiler adds capability bounds on stack variables in an LLVM intermediate representation (IR)-level pass just before instruction selection. This pass originally replaced all uses of LLVM `alloca` instructions with a call to `llvm.cheri.cap.bounds.set` with the appropriate size. While this ensures all stack variables are correctly bounded, this had at least three downsides. Firstly, it resulted in increased register pressure (and therefore additional stack spills) as the bounded capability was kept live from function entry to the last use. Secondly, it prevented reuse of stack slots for temporally distinct allocations. Finally, the compiler knows that it can re-materialize the address of stack slots, but by using an opaque intrinsic this was no longer possible.

Avoid bounded stack variables If a stack variable is only used for statically in-bounds loads and stores inside a single function, and the address is never taken, we can avoid adding the bounds since we know that the bounds checks will always succeed. In these cases, we can perform these loads and stores relative to the stack capability `$csp` and fold the offset into the immediate operand. This avoids creating the bounded stack capability and therefore massively reduces register pressure. Besides omitting a `CSetBounds` instruction, it also means that the compiler can re-materialize the pointer before uses and no longer needs to spill. This optimization does not weaken any security guarantees provided by CHERI. For cases where the `alloca` value is passed to another function or if it is used in any `getelementptr` instructions that cannot statically be proven to be in bounds, we still add the bounds. We currently perform a very conservative analysis in this pass and fall back to creating a bounded capability for almost every instruction other than in-bounds loads and stores.

[41] In a merged register file architecture, existing integer operations could be used for `uintptr_t` values.

[42] In load and store instructions, register number zero refers to `$ddc` since loading from a `NULL` pointer makes no sense and will always trap.

Re-materializing bounded stack variables In LLVM, all accesses relative to the stack pointer (represented as a `FrameIndex` internally) can generally be re-materialized anywhere in the function. This was broken by adding an intrinsic call since all references to the `alloca` instruction now refer to this `CSetBounds` instruction that cannot be re-materialized and therefore results in register spills at the start of the function. To avoid additional register spills, I added a new intrinsic `llvm.cheri.bounded.stack.cap` to LLVM and told LLVM that the intrinsic can be rematerialized at any time. For rematerialization, we also have to pretend that it is as cheap as a move instruction. This is not quite true as it is two instructions, but it is still cheaper than a stack spill.

Moving the intrinsic call closer to its use Originally, all uses of the stack variable use either `$csp` if bounds are not needed or they referred to the capability-bounding intrinsic. However, if we duplicate the intrinsic and move the call closer to the use, it means that the register holding the value is live for a shorter time and therefore it is more likely that the compiler will use a temporary register and avoid a spill to the stack. However, if we were to duplicate intrinsics with many uses, the additional `CSetBounds` instructions may be more expensive than one stack spill. Therefore, we do not duplicate the intrinsics by default if more than five uses exist.

Optimization effectiveness After adding these optimizations, the stack-frame size for `JSC::Lexer::lex()` went from 3264 bytes to 672 bytes.[43] Importantly, this is the same size as when we completely disable bounding of stack variables. Additionally, the size of the function changed from 28426 to 27583 instructions, i.e. 3.0% fewer instructions. Nevertheless, there is still a lot of potential for future optimizations since the MIPS stack-frame size for this function is only 288 bytes (but uses comparable 27800 instructions). Even If we assume that all values on the stack are pointers (which they are not), CHERI capabilities should only double this size to 576 bytes. Yet despite these code-generation inefficiencies in the CHERI compiler, pure-capability code performs very similarly to MIPS code (see Section 6.2 for benchmarks and analyses).

3.9 Unsupported C/C++ programming idioms

Although the goal for CHERI pure-capability C/C++ is to support as much existing code out-of-the-box as possible, there are some idioms, patterns and optimizations that cannot sensibly be supported in a pure-capability environment. It is important to note that the cases that we have found so far are all undefined or implementation-defined behaviour

[43] The current default parameters for these optimizations are: only set bounds if needed, allow rematerializing the value and avoid reusing the same intrinsic if more than five uses exist.

according to the C standard [147].[44] Moreover, we believe that these are rarely used and/or can easily be converted to CHERI-compatible code.

3.9.1 XOR-linked lists

In an architecture that uses integer pointers, it is possible to implement a doubly-linked list using only one pointer by using an exclusive-OR between the previous list element pointer and the next pointer [212]. This optimization is not compatible with CHERI capabilities, as the XOR operation would make the pointer be out-of-bounds and therefore non-dereferenceable. However, if one were to use capabilities with very large bounds, i.e. ones that span all list elements, this optimization would still be possible. Alternatively, the list could be implemented using virtual addresses and prior to dereferencing, a capability could be derived from a large ambient one. However, we do not believe it is a good idea as it prevents enforcing spatial safety and is therefore not possible by default.

3.9.2 Offsets relative to the current structure

We noticed that some code assumes that adding a difference between two objects to one of the pointers yields a valid, dereferenceable pointer. As this creates an out-of-bounds pointer, it is not supported by the C standard [112, §6.5.6.8] or PVI provenance semantics [147]. In CHERI pure-capability mode this pattern will result in a bounds violation or an unrepresentable capability.

One interesting real-world case of this pattern was found in the QtBase library. It was used in the Qt string class, `QString`, to avoid relocation processing at start-up [83]. Instead of storing a pointer to the data, the `QString` implementation stores an offset relative to the current `this` pointer.[45] For global data and newly allocated `QString`s the string data immediately follows the `QString` and is part of the same allocation. In this case adding the offset to `this` works even in pure-capability mode. However, `QString` can also be used to create a non-owning reference to other string data. Adding the difference between the external string data and the `this` pointer results in out-of-bounds values and run-time crashes. I fixed this — while retaining the relocation optimization — by storing an offset for contiguous data, and a capability for non-owning external data references.

There is no viable approach to support this pattern with CHERI capabilities, and it relies on undefined behaviour, so we do not support it in pure-capability mode.

[44] The case of byte-wise copying (see Section 3.9.4) is permitted as long as the optimization of word-by-word copies is not employed [147].

[45] For global data this offset is a static-link-time constant and (unlike pointers) does not require any run-time relocation processing.

3.9.3 Updating pointers after `realloc()`

Some programs grow data structures containing pointers using `realloc()`. To update pointers within this data structure they then add the difference between the old structure pointer and the return value to every pointer. We saw this, for example, in one of the SPEC benchmarks, `429.mcf`. While this will succeed on most architectures using integer pointers, it is undefined behaviour [112, §6.5.6.8] and not guaranteed to work [53, 147]. For CHERI, this pattern will not be supported, and we require re-derivation of a new capability from the `realloc()` return value (see '*CheriABI*' [53] for more details).

3.9.4 Byte-wise copying of data

Copying data byte-by-byte (or with a larger, non-capability-aligned granularity) will not preserve tag bits. This is one case where pure-capability CHERI differs from the PVI provenance model for C [147]. After a byte-wise copy, the result will be a bitwise identical copy of the capability that cannot be dereferenced due to the missing tag bit. These byte-by-byte (or word-by-word) copies happen in `libc` functions such as `memcpy()` and `memmove()` but should generally be less common in higher level programs. One other location where we found this to be a problem was `qsort()` which performs word-size swaps during sorting. We also discovered that `PostgreSQL` was using a modified copy of the FreeBSD `qsort()` code that exhibited the same behaviour. However, most of these functions are in low-level system libraries and fixing them once should allow all user code to work as expected. So far, we have rarely seen this problem while porting various programs to pure-capability CHERI. Moreover, the compiler already transforms loops that look like `memcpy()` to a call if it is deemed beneficial, so at higher optimization levels these byte-wise copies might actually be turned into a tag-preserving `memcpy()` call.[46]

3.9.5 Using high pointer bits

On 64-bit architectures the high bits of a pointer are often unused as most CPUs only support 48–52 bits of virtual address space. By masking prior to dereferencing,[47] some projects (e.g. jemalloc [72]) store additional data in the (architecturally unused) high bits. Modifying these address bits of a CHERI capability can cause it to become unrepresentable (which results in the tag bit getting cleared) so this trick no longer works with compressed CHERI capabilities. However, CHERI capabilities allow storing at least nine bits (potentially more depending on alignment) in the *offset* field, so there should be less

[46] It is unclear whether the compiler should be allowed to optimize a programmer-written loop that copies byte-by-byte (and therefore does not maintain tags) into a `memcpy()` that maintains tags. Our current inclination is that existing code will expect tags to be maintained, but a CHERI-aware programmer might expect them to be cleared. This could be similar to `memset()` vs. `explicit_bzero()` concerns where the compiler is sometimes 'too smart'.

[47] Some architectures (e.g. AArch64 [12, §12.5.1]) also provide a hardware mechanism to ignore high bits of pointers.

need to (ab)use the high pointer bits in pure-capability CHERI. In the future, we may decide to ignore the high address bits of capabilities for bounds and use them for other purposes instead.

3.9.6 Yet to be discovered issues

In the process of porting many programs and libraries to CHERI (including the multi-million line codebase of WebKit) we have not yet seen any other idioms that are fundamentally incompatible with CHERI pure-capability C. However, it is certainly possible that code exists which would require significant re-engineering in order to run in a pure-capability CHERI environment.

3.10 Future changes to pure-capability C/C++

Over the course of my PhD I have made various changes to improve the C-compatibility of pure-capability CHERI code, the most notable being the *address* interpretation of capabilities. Nevertheless, we have since discovered problems and opportunities that may best be addressed by further changes to the compilation model.

3.10.1 True offset semantics

As noted in Section 3.6, the current offset interpretation of CHERI capabilities is inconsistent. Therefore, a future change to the *offset* compilation mode could return the capability offset in *all* cases (including pointer to integer casts and `switch` statements). This would remove these inconsistencies and further reduce leakage of virtual addresses, hereby improving compatibility with a potential C implementation that uses copying garbage collection.

3.10.2 Explicit provenance source for `uintptr_t`

Currently, arithmetic operations on `uintptr_t` always inherit provenance from the left-hand-side. While this is a simple rule (and also easy to implement in the compiler), it results in surprising compatibility problems where `constant_int+int_or_ptr` always creates an untagged value, even though it could reasonably be expected to behave in the same way as `int_or_ptr+constant_int`. This problem could be fixed by requiring an explicit source of provenance in every `uintptr_t` arithmetic operation. One way of achieving this would be to cast all other operands to a non-provenance carrying integer type.[48] For cases where this is not possible, we could introduce a `cheri_provenance_source` annotation. Initially, this would warn in cases where the provenance source is ambiguous and select the

[48] While integer arguments are automatically promoted to `uintptr_t` in the expression, we can detect this in the compiler and avoid unnecessary annotations for these cases.

left-hand-side. If we discover that this warning is only triggered rarely, we could promote it to be an error by default.

3.10.3 Strict compilation mode (CHERI sanitizer)

Recently, AddressSanitizer (ASan) added support for finding invalid pointer comparison according to the C standard, i.e. comparisons between distinct objects [148], using `-fsanitize=pointer-comparison`. Without sub-object bounds we could implement this trivially in CHERI; with sub-object bounds we need to do the same thing as ASan and ask `malloc()` for the containing allocation. However, we could still use a CHERI capability subset test to provide a fast path. We could also add support for detecting pointer subtractions between distinct objects using the same approach. Finally, we could also use CHERI bounds to detect the creation of out-of-bounds pointers that are not one-past-the-end. Such a strict compilation mode could be added to extend UndefinedBehaviorSanitizer (UBSan) with a `-fsanitize=cheri` flag to provide faster checks.

3.11 Summary

In this chapter I have presented some key differences between conventional architectures and CHERI pure-capability C/C++. A key lesson learned is that while the original pure-capability model of using capabilities offsets improves C compatibility compared to the original CHERI implementation, it introduced other new incompatibilities due to the design goal of being able to support copying garbage collection in C. These incompatibilities have been dealt with by introducing the *address interpretation* of capabilities and by making it the default pure-capability compilation mode. While incompatibilities still remain even with the new refined pure-capability C/C++ semantics, most of these rely on undefined or implementation-defined behaviour. Nevertheless, these cases are common in practice and we should not restrict pure-capability C/C++ to support an idealized standard but instead focus on real-world code to minimize the cost of adoption.

4 PURE-CAPABILITY CHERI LINKAGE

This chapter focuses on the challenges and opportunities that pure-capability CHERI brings to static and dynamic linking for C (and C++) programs running on a POSIX operating system (in this case CheriBSD). While not explicitly part of the C or C++ standards, dynamic linking is an essential part of real-world software in all widely used desktop, server and mobile platforms. Dynamic linking generally results in less efficient generated code, yet memory conservation due to not duplicating multiple copies of the same code and data can improve performance. More importantly, upgrading one shared library provides bug and security fixes for all programs using the library without having to recompile and redistribute.

In this chapter, I present program linkage design trade-offs and resulting opportunities for strong memory protection and compartmentalization. Reducing the amount of privilege (i.e. accessible memory) available from linkage-derived implicit pointers can limit the impact of bugs or potential attacks. When using CHERI, we can limit this privilege by shrinking the bounds on capabilities that are used to reference global variables and functions. Section 4.2 presents various approaches of obtaining these capabilities and introduces the existing pure-capability CHERI linkage model[1] implementations with different performance and privilege-minimization properties. For this dissertation I implemented two models: one is designed to yield a fair performance comparison to the MIPS baseline; the other focuses on bounds minimization. The use of CHERI capabilities instead of integer addresses imposes certain restrictions on the implementation, so I present the challenges arising from this change and the chosen solutions in Sections 4.3 to 4.5. These challenges cover essential parts of program execution such as function calls across library boundaries, lazy function resolution, initialization of global pointers and correct handling of C/C++ function pointers. Next, in Section 4.6, I show how privilege can further be reduced using Global Visibility Enforcement (GVE), a novel technique that ensures least-privilege globals access: every function can only access the functions and global variables that are used in the source code. In order to improve security and performance of pure-capability linkage, I also added new features to the CHERI ISA (see Section 4.7). The most notable change is the introduction of *sentry* capabilities, which can be used to provide code and data isolation between libraries at no additional cost. *Sentry* capabilities also allow basic compartmentalization

[1]Throughout this chapter, the term 'linkage model' is equivalent to application binary interface (ABI); it is used to highlight that the only notable differences between the presented ABIs arise from the behaviour of the static and dynamic linker. For example, all pure-capability ABIs use the same relocations, executable format and partitioning of caller-/callee-saved registers; calling conventions are also almost identical.

to be added at naturally occurring boundaries (such as libraries, compilation units and even individual functions) without making any changes to the application and library source code. Importantly, the extent of privilege reduction and compartmentalization can be adjusted both at run time and during compilation. Finally, I evaluate performance, compatibility and privilege minimization properties of pure-capability linkage in Section 4.8 and propose future changes in Section 4.9.

I presented a portion of this work at the peer-reviewed PRiSC 2019 workshop [188] and my implementation of pure-capability dynamic linking also contributed to the ASPLOS 2019 paper '*CheriABI*' [53, 54].

4.1 Introduction

Pure-capability CHERI linkage is similar to linkage on conventional architectures (see Section 2.3). However, in order to provide improved security and isolation, we must make some adjustments and revisit design choices. Both the static and dynamic linkers play a significant role in refining capabilities available to the program at run time. This includes language visible pointers (such as pointer-type global variables and function pointers) as well as implicit pointers invisible to the programmer (such as those used for global variable accesses and function calls). Choices in linkage can also affect which memory regions are accessible to the currently executing code. From a security point of view, these language-invisible pointers are especially important since they are often modified/corrupted during the process of obtaining arbitrary code execution (e.g. in return-oriented programming (ROP) attacks) [221].

The CHERI architecture enforces provenance validity, integrity and monotonicity on all capabilities (see Section 2.2). When using capabilities to implement all explicit and implicit pointers in a C/C++ runtime environment, this imposes certain challenges upon the implementation that are not present with integer pointers. Certain common strategies are either impossible, more difficult or less efficient to use. This includes cases like using an integer constant as a pointer or deriving data pointers from code pointers.[2]

All the choices that I have made when designing CHERI pure-capability linkage are guided by two design principles. Firstly, whenever possible, I follow *the principle of least privilege* [194, 195] since this limits the impact a potential attacker can have. For example, a dynamically linked program does not require direct access to memory assigned to libc.so,

[2]The former is already impossible in position-independent code (as required for address-space layout randomization (ASLR)), but the latter has, perhaps sadly, been popularized by the presence of ASLR.

yet in contemporary architectures it is accessible.[3] Secondly, within any given system architecture, components with distinct levels of trust and differing need for protection exist. Therefore, it should be possible to use security-critical libraries (e.g. SSL/private key processing) without exposing their internals to more performance-oriented (and often spatially unsafe) code such as image decoding. This should be possible within the same process without having to use a uniform call security policy. Another example is `malloc()`, which may want to protect itself from the caller (e.g. to avoid leaking capabilities to distinct allocations). However, the caller generally trusts `malloc()` and should not need to pay the cost of isolation.

I therefore propose, prototype, and evaluate two pure-capability dynamic linkage designs with different design choices: one strictly follows these principles, while the other is designed to have performance that is comparable to the MIPS baseline. Furthermore, I explore the following hypotheses:

- CHERI linkage can completely eliminate out-of-bounds memory accesses for global variables.
- Different ABIs can offer different security guarantees, and they can interoperate within the same process.
- Accessible memory can be substantially reduced, while still being able to run all C/C++ programs. For example, memory outside the current dynamic shared object (DSO) should be inaccessible (except for exported symbols)
- We can reduce the size of the trusted computing base (TCB): only the run-time linker and kernel must be fully trusted, but not system libraries such as libc. Due to CHERI's architectural properties, we also require less trust in the compiler and linker, but handling malicious ELF files [61] is not in scope for this dissertation.
- We can provide strong protection against control-flow hijacking by limiting the available of jump targets. CHERI hardware already prevents arbitrary jumps, but if external code pointers span the whole DSO, all code within it is reachable. By using *sentry* capabilities, we can ensure that all jumps target valid entry points and thus CHERI provides a form of Control-flow Integrity (CFI).

4.2 CHERI pure-capability linkage models

C source code contains explicit pointers (e.g. `char*` function parameters) as well as implicit, compiler-generated pointers. Consider for example Listing 4.1: a simple function that passes the value of a global variable `global_var` to a function `external_function()`. This sample program does not contain any pointers at the C level. However, similar to

[3] It may not be trivial to access this memory, considering that ASLR is usually enabled and therefore the location of libc.so is not known. However, the memory is still accessible once the correct address has been determined, so ASLR-bypassing exploit chains always begin with an information disclosure. For CHERI linkage we would like to avoid any kind of *probabilistic* defence and instead ensure unnecessary privilege is actually not available.

```
extern long external_function(long arg);

long global_var;

long foo() {
   return external_function(global_var);
}
```

Listing 4.1: Sample C source code that loads a global variable and calls a function

C++ virtual function calls relying on compiler-generated *vtable* pointers, all accesses to global variables also rely on *implicit, language-invisible* pointers such as those contained in a global offset table (GOT). Moreover, the call to `external_function()` requires the program to synthesize a pointer to the target function to transfer control flow and a mechanism to return to the caller. The latter is usually implemented with a return pointer (or link register) that is often stored on the stack. These implicit, language invisible pointers are an attractive target for attackers as they usually reside on the stack and when corrupted can be used to subvert control flow (e.g. using a ROP attack). Therefore, many security mitigation strategies exist to protect the return pointer [46, 108, 227, 251], and the chosen CHERI linkage model should also ensure this protection. While CHERI already raises the bar for code injections by providing spatial and referential safety for language-visible pointers (see Chapter 3), the linkage model must participate in robust protection. A naïve implementation of pure-capability linkage that does not reduce bounds or permissions on implied pointers (e.g. call targets) would grant attackers capabilities that could be used in an exploit. A particularly important capability is the default data capability (`$ddc`), which is implicitly used as the base capability for the existing MIPS loads and stores that use an integer address. Setting this capability to `NULL` ensures that all memory accesse that do not explicitly use capability will trap at run time.

Several ways exist in which these implicit pointers to global variables or functions can be generated [137, chapters 7&8], with each having different implications for bounds-minimization and performance. In the remainder of this chapter I will refer to both global variables and implicit pointers to functions as *globals*. Moreover, it is possible to use different access methods depending on the type of global. To generate the address of a global variable, the approach is generally either to create a pointer directly to the global variable or to indirect via a table (e.g. the GOT) to obtain the actual pointer. To generate the address of the target function, we can use the same approach as for variables (usually loading from a table). For functions that are known to be within the same DSO (or if the offset between current and target function is known), it is also possible to use a relative branch-and-link instruction that jumps to a fixed offset from the current program counter. However, this approach is at odds with the goal of using tight bounds for all capabilities since the program counter capability (`$pcc`) must also grant access to the target function.

To minimize privilege in pure-capability CHERI linkage, we must not only compute the correct address but also set appropriate bounds and permissions for implied pointers

to globals. I consider the following five approaches for accessing globals in CHERI pure-capability linkage:

Synthesizing fixed pointer values Position-dependent code knows the run-time address of the target function/variable at static link time and can synthesize this fixed pointer (either by embedding it in the instruction stream or loading from a constant data section). This approach works very well in architectures such as x86 where even 64-bit constants can be included in a single instruction. However, due to the monotonicity requirements imposed by CHERI capabilities, this is not a workable solution for pure-capability code. It would require deriving all globals from an ambient capability that spans all code and data memory and holding this throughout program execution. This approach is clearly not ideal from a privilege-reduction point of view. Furthermore, it cannot be used when building shared libraries that may be loaded at any address at run time. Therefore, we do not further consider this approach for loading globals.

Relative addressing from a base register For position-independent code, it is common to derive the addresses of globals from another register that uniquely identifies the current code domain. The natural choice for this is using the program counter register and generating the addresses of global variables and functions by adding a constant offset to it. This approach requires the offset between code and data segments to be known. It is used for example by RISC-V [190], AArch64 [11] and x86_64 [145], and all these architectures have added instructions to efficiently encode addressing relative to the program counter.[4]

Base register relative indirection A slight variation of the previous scheme can be used to access globals whose relative offset to the base register is not known. Instead of accessing the global directly, the address of the GOT is derived from the base register and the global address is loaded from this table. This scheme is used by the MIPS n64 ABI [180] (see Section 4.2.1) for all globals (even those with a known offset). It could also be used on CHERI (if the bounds of the program counter span the entire DSO), and a variation of this scheme is one of the linkage models that I have implemented (see Section 4.2.4).

Using an ABI-reserved register for globals Another option to obtain the location of globals is to reserve a register in the ABI that holds the base location of all globals (or more commonly a table with addresses) and must be set correctly on function entry. For example, the PPC ELFv1 ABI uses this approach and one register (`$r2`) is a dedicated pointer to a table containing addresses of all global variables [223, 261]. When using this approach, one register cannot be used for other purposes and may need to be saved on

[4]We do not have `$pc`-relative load/store instructions in CHERI-MIPS since MIPS only added them in 2014 (MIPSr6) and CHERI is based on an older version to avoid patent issues.

ABI name	Bounds for Code	Bounds for Data	Access to globals pointer	Function-pointer representation
MIPS n64 (baseline)	✗	✗	relative to `$pc`	target function
Legacy (obsolete)	✗	(✓)	relative to `$pc`	target function
Function-descriptor prototype	function	✓	set on entry	function descriptor
PC-relative ABI	DSO	✓	relative to `$pcc`	target function
PLT ABI	function	✓	set on entry	trampoline

Table 4.1: Comparison of the different linkage models.

every function call. Moreover, it requires all *callers* to set up the correct globals pointer[5] which can add unnecessary overhead if the called function does not use any globals. I also implemented a linkage model based on this approach (see Section 4.2.5) where every function requires a `$cgp` (capability globals pointer) register to be set on function entry.

Asking a more privileged component Finally, it is also possible to ask a more privileged component (e.g. the run-time linker or the kernel) for the correct table of globals. While this avoids the need for a special register, it adds another function call/domain transition every time a function needs access to globals.[6] This approach is therefore too costly for the general global variable/function pointer addressing. However, it is sometimes used for thread-local storage (TLS) since accesses to thread-local variables are less common than accessing globals. Such a model could work on CHERI for general global access, but due to the expected performance issues I did not implement it. Nevertheless, it is used for TLS, and Section 4.7.3 gives a brief overview of that mechanism.

In the following subsections I present the different linkage models that can be used on the CHERI-MIPS platform. A summary of the different ABIs presented in this section can be seen in Table 4.1. The following subsections includes assembly listings that assume knowledge on register usage, specific instructions and calling conventions. I provide higher-level explanations for each of these listings but readers may also wish to consult Appendix A for further background. This appendix provides a brief overview of the MIPS and pure-capability CHERI register usage as well as descriptions of CHERI instructions used throughout this dissertation.

4.2.1 MIPS n64 baseline

The architecture that our current CHERI field-programmable gate array (FPGA) prototype runs on is derived from the 64-bit MIPS4 ISA and uses the n64 ABI [180] for MIPS code. To access global variables and call functions, the MIPS n64 ABI loads all variables and

[5]This can often be omitted for local function calls (within the same DSO) since these generally share the same globals pointer.

[6]These accesses could potentially include computing addresses of other functions. Therefore, the mechanism to invoke the privileged component cannot use the normal function call sequence.

```
1 lui $1, %hi(%neg(%gp_rel(foo)))
2 daddu $1, $1, $25
3 daddiu $gp, $1, %lo(%neg(%gp_rel(foo)))
4 ld $1, %got_disp(global_var)($gp)
5 ld $a0, 0($1)
6 ld $t9, %call16(external_function)($gp)
7 jalr $25
8 nop
```

compute globals pointer `$gp` from entry point register `$t9` by adding a constant offset

load `global_var` from GOT into argument register `$a0`

load address of `external_func` into `$t9` and transfer control flow (the `nop` is used to fill branch-delay slot)

Listing 4.2: The example code from Listing 4.1 compiled for the MIPS n64 ABI.

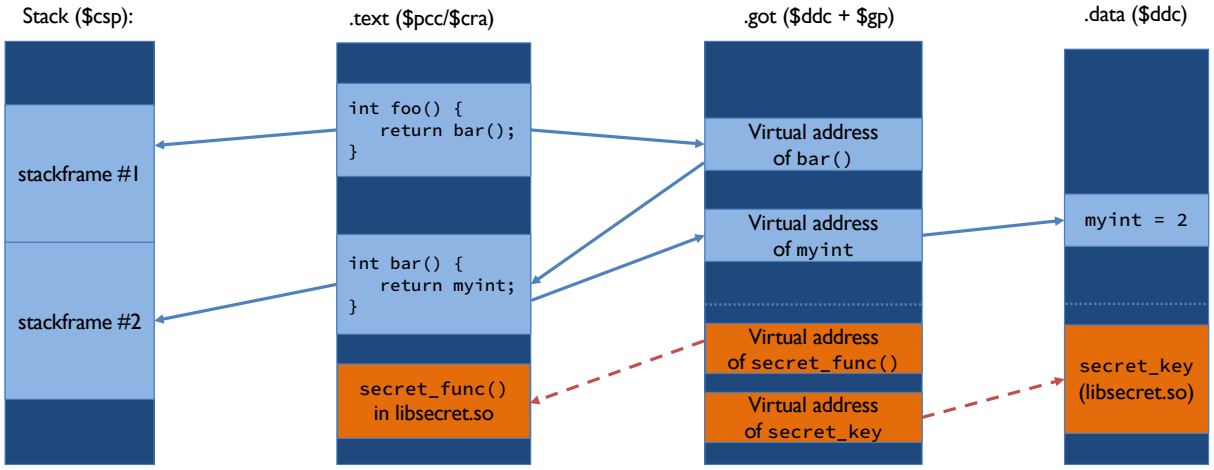


Figure 4.1: Original linkage model using virtual addresses and the MIPS GOT. Since this model does not use capabilities, all memory is accessible (dark blue in this diagram) unless the MMU has been configured to prevent those accesses. Due to using full-address-space capabilities for `$ddc` and `$pcc`, memory that should be inaccessible (highlighted in orange) is reachable.

function pointers from the global offset table (GOT).[7] On function entry the address of the GOT is computed and stored in the callee-save `$gp` register. This globals pointer (`$gp`) refers to an offset within GOT that allows fast access to all globals used in the current function.[8] Listing 4.2 shows the subset of the generated assembly code that loads the global variable and calls another function. The n64 ABI mandates that on entry `$t9` (`$25` in the listing) should hold the start address of the function [105], and this is used to compute `$gp`. This pointer to the GOT is then used to load `global_var` and the address of `external_function`. When performing function calls, the jump must always be performed via `$t9` so that the callee can derive `$gp` from `$t9` (since originally MIPS did not include `$pc`-relative instructions).

```
1  cgetoffset $t9, $c12
2  lui $1, %hi(%neg(%gp_rel(foo)))
3  daddu $1, $1, $25
4  daddiu $gp, $1, %lo(%neg(%gp_rel(foo)))
5  ld $1, %got_disp(.size.global_var)($gp)
6  ld $2, %got_disp(global_var)($gp)
7  ld $1, 0($1)
8  cfromddc $c1, $2
9  csetbounds $c1, $c1, $1
10 cld $a0, $zero, 0($c1)
11 ld $1, %call16(external_function)($gp)
12 cgetpccsetoffset $c12, $1
13 cjalr $c12, $c17
14 nop
```

compute virtual address of GOT in register `$gp` using the entry point register `$c12` and adding a constant offset

load the value of `global_var` by
① reading the virtual address from the GOT
② using this to create a capability from `$ddc`
③ loading the size of the global variable from `.size.global_var`
④ setting the capability bounds to this size
⑤ loading the value into argument register `$a0` (`$4`)

load virtual address of `external_func` from the GOT, derive the (unbounded) target capability into `$c12` from `$pcc` using `cgetpccsetoffset` and transfer control flow

Listing 4.3: The code from Listing 4.1 compiled for the original CHERI linkage model.

4.2.2 Original CHERI static linkage model

Prior to the start of my PhD, the CHERI pure-capability compilation mode used a rather insecure and inefficient linkage model. While this model was only a stepping-stone before real linker work, it is useful to review it as the starting point. All capabilities for global variables are created by reading the virtual address from the MIPS GOT and deriving a capability to that address from `$ddc` using `cfromddc`. To bound those capabilities, we load the size of that variable from a special symbol `.size.<symbolname>`. Using this size information, the compiler then emits a `CSetBounds` instruction so that pointers to globals do not grant access to the full address space. As can be seen in Listing 4.3, this model reuses the existing MIPS n64 `$gp` register and computes it from the CHERI equivalent of `$t9`, the entry point capability `$c12`. Code pointers use a slightly different code sequence as they need to be executable and therefore derive from `$pcc` using `cgetpccsetoffset`.[9] Additionally, we do not set bounds on code pointers since we need a full-address-space `$pcc` to derive jump targets for calls.

We were able to run most statically linked programs using this linkage model. However, when we started to use dynamic linking, we had to add many workarounds inside the run-time linker (RTLD). The `.size.<symbolname>` symbols that are filled in at static link time caused many issues, and we had to add workarounds in RTLD to overwrite these values at load-time. Not only are these workarounds quite fragile, they also add start-up overhead due to symbol lookup. Moreover, using this linkage model is also inefficient since it adds additional instructions to every global variable access and function call (see

[7] Some small data objects may be placed directly in the GOT to avoid one indirection.

[8] Due to the 16-bit immediate range of the MIPS load instructions, the linker arranges multiple 64K GOTs such that every function can access its required globals within one block (it may duplicate some entries to achieve this) [15]. An alternative to this *multi-GOT* scheme is to use an instruction sequence to generate a larger immediate. This can be enabled using the `-mxgot` compiler flag.

[9] As the GOT contains virtual addresses this should probably be `cgetpccsetaddress`, but `$pcc` must be a full-address-space capability (i.e. with offset zero), making setting the offset and setting the virtual address equivalent. Moreover, this model was developed while the CHERI pure-capability ABI was still focused on the offset interpretation of capabilities so using the offset was the natural choice (see Section 3.6). Furthermore, using the offset made it easier to support code running in a libcheri compartment [248] with a non-zero base `$pcc`, as changing `$pcc` can be used to relocate code running at 'virtual address' zero.

Section 4.8.3). It also increases the sizes of binaries more than is necessary: code size increases due to the inefficient code generation and the dynamic symbol table needing to include the size symbols.

Most importantly, this linkage model clearly grants too much ambient privilege. The memory regions that must be accessible for this linkage model can be seen in Figure 4.1. While pointers to global variables will be correctly bounded at run time, every executing function still has full-address-space `$ddc` and `$pcc` values that grant enough ambient privilege to circumvent any bounds that have been set on globals.

4.2.3 Function-descriptor prototype

In the original linkage model, each use of a global needed to (re-)derive a capability from the ambient `$ddc` or `$pcc`. Our first step towards dynamic linkage instead generated a table once at load-time, the `captable`. This model uses the *reserved register for globals pointer* approach to access globals. On function entry, register `$c14` points to a capability equivalent of the MIPS GOT. This table, hereafter referred to as `captable`, contains bounded capabilities for every global instead of just the virtual address, as is the case with the GOT. Within the same DSO, the globals pointer remains the same, but when calling a function in a different DSO the correct value must be loaded via a procedure linkage table (PLT) stub. In the case of calling a C function pointer, the function pointer is not an executable capability but instead it is a *function descriptor* (a pair of capabilities, one for the target function and one for the target `$c14` value). This is not specific to CHERI, function descriptors are also used by PPC (in the ELFv1 ABI) [223], PA-RISC [102] and Itanium [109].

Jessica Clarke created this prototype in 2017 and it had strong capability bounds minimization properties[10] [42]. However, there were two downsides to this model that resulted in us not adopting it by default:

- We needed a linkage model that is comparable to the MIPS existing code-generation to be able to perform meaningful benchmarks. If we use a different model, all measurements of pure-capability overhead relative to a 64-bit MIPS ISA are skewed by the different numbers of instructions used for every global access (see Section 4.8.3).
- Code pointers can no longer be jumped to directly. To handle multiple different ABIs, the kernel must be modified to perform different actions based on whether the target code capability has execute permission or not (e.g. for the `sigaction()` implementation). This makes it harder to mix multiple different ABIs.

While we did not choose to adopt this linkage model by default, some of the design choices were reused for the linkage models I implemented later. For example, the use of a capability table for global variables instead of a GOT has been adopted and the code generated by the compiler is also similar in the PLT ABI (see Section 4.2.5).

[10]The bounds reduction is the same as in the PLT model that I implemented later (see Section 4.2.5).

```
1 cmove $c18, $c14                                  } save live-in captable pointer $c14 in callee-save $c18
2 clc $c1, $0, %mctdata(global_var)($c14)           } load global_var into argument register $a0 using a tightly
3 clw $1, $0, 0($c1)                                } bounded capability from the captable
4 clc $c12, $0, %mctcall(external_function)($c14)   } load capability for external_func into $c12 and transfer control
5 cjalr $c12, $c17                                  } flow
6 nop                                               }
7 cmove $c14, $c18                                  } restore $c14 value after call to external_func
```

Listing 4.4: The code from Listing 4.1 compiled for the function-descriptor prototype [42].

4.2.4 PC-relative ABI

In many contemporary architectures a common idiom when running position independent code is to perform loads and stores relative to the program counter. While this would be a *possible* approach in the world of CHERI, it would not minimize privilege as it would require the program counter capability to have read, write and execute permissions (since it must be possible to derive all capabilities from `$pcc`). Furthermore, we could use only existing MMU-based protection to guard the program code from malicious modification rather than capability bounds and permissions. Instead of loading and storing directly via the program counter, I implemented a variation of a *PC-relative ABI* that allows using a read-only `$pcc`. In this ABI, `$pcc` spans the whole text segment and part of the read-only after relocation processing (RELRO) segment — at least the part containing the `captable`.

This model is slightly different from a traditional PC-relative ABI, in that global variables are never accessed directly via `$pcc`.[11] Instead, a capability to the global must be obtained by indexing into the DSO's `captable`. The `captable` is located at a statically known offset from `$pcc`, so whenever a function accesses globals, we must first derive the pointer to the `captable` from the program counter. In terms of code-generation this model is almost identical to the MIPS n64 model: the pointer to the globals (`$cgp`) is computed by adding a link-time constant (the difference between the current program counter and the location of the `captable`) to the program counter capability `$pcc`.

Capabilities for global variables and function calls can be loaded directly from the `captable` using the assembler expressions `%captab20`/`%capcall20`, which the static linker replaces with the correct index into the `captable`. The run-time linker ensures that these capabilities have the appropriate bounds and permissions (see Section 4.4). Example code generated for this ABI can be seen in Listing 4.5.

Advantages While the *PC-relative ABI* is not the ideal choice from a privilege minimization point of view, it does have some important advantages. First, the available ambient privilege has been significantly reduced (see Figure 4.2). All global variables have tight bounds since they are read from the `captable` instead of being derived from `$ddc` (now

[11] Even in conventional architectures, external references are indirected, and only DSO-local references are made via the program counter.

```
1 lui $1, %pcrel_hi(_CHERI_CAPABILITY_TABLE_-8)
2 daddiu $1, $1, %pcrel_lo(_CHERI_CAPABILITY_TABLE_-4)
3 cgetpccincoffset $c1, $1
4 clcbi $c2, %captab20(global_var)($c1)
5 cld $a0, $zero, 0($c2)
6 clcbi $c12, %capcall20(external_function)($c1)
7 cjalr $c12, $c17
8 nop
```

derive capability to `captable` from from program counter register `$pcc` by adding a constant offset

load `global_var` into argument register `$a0` using a tightly bounded capability from the `captable`

load capability for `external_func` into `$c12` and transfer control flow

Listing 4.5: The code from Listing 4.1 compiled for the PC-relative linkage model

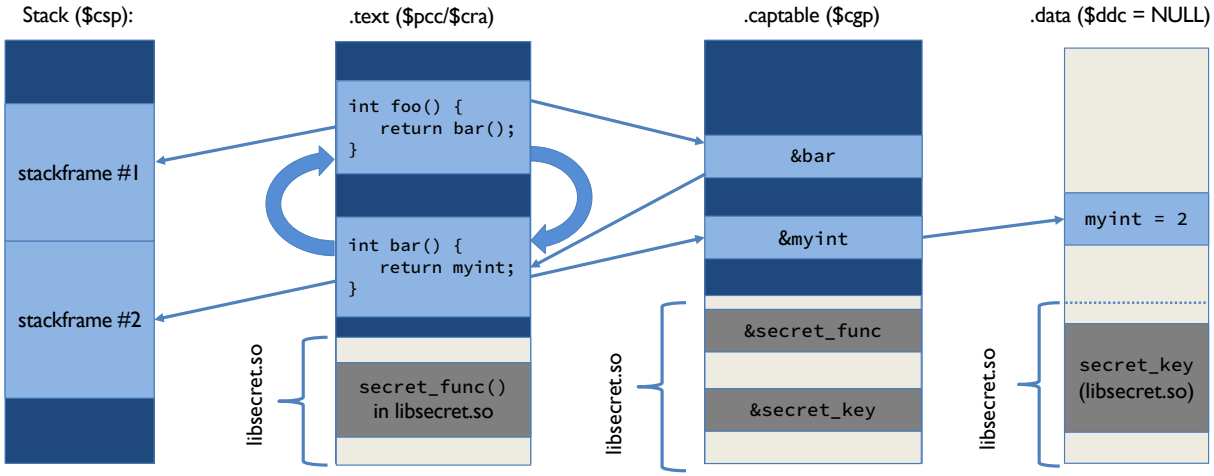


Figure 4.2: PC-relative linkage model. Only the code segment of the current DSO is accessible whereas other DSOs cannot be reached (highlighted in grey). Moreover, in this model `$ddc` is `NULL` so only globals listed in the `captable` are accessible.

unused and set to `NULL`). Additionally, all code pointers can be bounded to the current library instead of spanning the entire address space.

Secondly, `$cgp` can be derived from `$pcc`, thereby avoiding the need for function descriptors or trampolines when crossing library boundaries. This also avoids complexity with C-language function pointers (see Section 4.5) and ensures that function calls to external libraries only require PLT stubs if lazy binding is enabled (see Section 4.3.3). This is especially important when invoking function-pointer callbacks (e.g. in `qsort()`) since the function will be executed from a completely different context.

Thirdly, the sample code compiles to the same number of instructions and the same amount of memory accesses as in the MIPS n64 ABI (see Listing 4.5). Therefore, the only performance difference when accessing global variables or loading jump targets can be attributed to the larger cache-pressure. This is caused by doubling the size of table entries in the transition from 64-bit offsets in the GOT to 128-bit (or even 256-bit in the uncompressed CHERI implementation) capabilities in the `captable`.

Finally, jumps to local functions could be performed without requiring a `captable` entry by adding a constant offset to the current `$pcc`. We chose not to do this to retain fair performance comparisons to the MIPS n64 baseline.[12]

[12]However, the RISC-V implementation of CHERI will use this model to match the RISC-V baseline.

Disadvantages However, there are also some disadvantages to using the PC-relative linkage model. Most importantly, the bounds of `$pcc` are larger than they need to be. Therefore, this model does not conform to the *principle of least privilege*. Additionally, code pointers grant access to all global variables (since `$cgp` — and therefore access to all entries in the table — can be obtained from `$pcc`). This should rarely matter for code within a single DSO since all these values are already accessible using the current `$cgp`.[13] However, the return capability `$cra` and all `captable` entries for external functions can grant access to the `$cgp` value for *other* DSOs. As this would allow an attacker with arbitrary code execution to recursively load any capability (and therefore be able to read sensitive data or jump to DSO-private functions); this is a serious flaw in the model. The proposed solution to this problem is a new hardware feature — *sentry* capabilities — which will be introduced in Section 4.7.1. Another constraint imposed by this model is that data must now reside at a constant offset from the code. This makes it harder to reuse the same code but with different data — as might be desirable when using sandboxes. Sometimes this can be worked around using virtual memory mapping tricks, but it makes this ABI more difficult to use in a single-address-space operating system or when sharing address spaces between different processes.[14]

Summary Overall, the PC-relative ABI allows a significant reduction in capability bounds while generating code that is perfectly comparable to the MIPS baseline for performance evaluations. We therefore chose to make this the default linkage choice in CheriBSD, but we also allow mixing it with other ABIs (see Section 4.3.2) to enable stronger protection for sensitive code.

4.2.5 PLT ABI

To further reduce the bounds on capabilities available, we must deviate from the *PC-relative* model. The other fully elaborated[15] linkage model I have implemented is the so-called *PLT ABI*. As noted earlier, this model is similar to the function-descriptor model. On entry, every function receives a capability to the `captable` in register `$cgp`. Unlike the function-descriptor model, I chose register `$c26` instead of `$c14`.[16] As all globals can be accessed using `$cgp`, we only need `$pcc` to execute the function's instructions. Therefore, we can now restrict the program counter bounds to include only the current function and remove all permissions other than execute (see Figure 4.3).

[13] Nevertheless, it may be desirable to reduce the number of jump targets and valid capabilities that are available to an attacker (see Section 4.6).

[14] CHERI makes the idea of doing this quite attractive and there is ongoing research in providing co-processes that share the same memory on top of a UNIX kernel.

[15] We are running regular continuous integration jobs that run the FreeBSD test suite using this ABI.

[16] Register `$c26` (also called `$idc`) is architecturally defined as the register into which `CCall` installs the data capability. I originally planned to use `CCall` to transition between DSOs so this choice avoids the need to move `$cgp` into another register. However, ultimately I did not end up using `CCall` but instead added a new hardware feature (see Section 4.7.1) to perform secure domain transition.

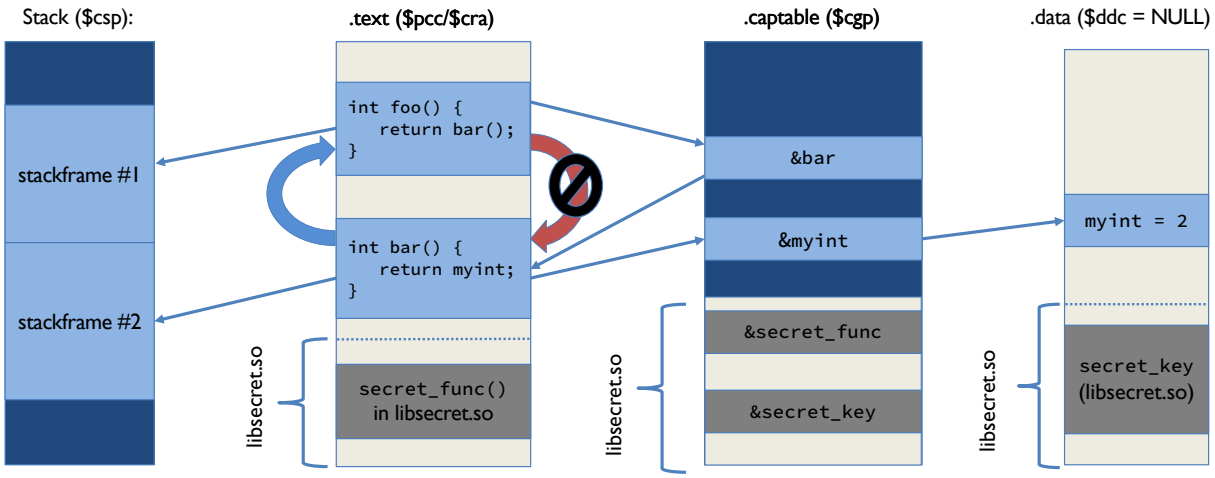


Figure 4.3: PLT linkage model with fine-grained bounds on functions. Unlike the PC-relative model, it is not possible to derive a capability to `bar()` from the `foo()` `$pcc`. However, it is still possible to return using register `$cra`.

```
1 cmove $c18, $c26                                  } save live-in captable pointer $cgp
2 clcbi $c1, %captab20(global_var)($c18)            } load global_var into argument register $a0 using a
3 cld $a0, $zero, 0($c1)                            } tightly bounded capability from the captable
4 clcbi $c12, %capcall20(external_function)($c18)   } load capability for external_func into $c12 and
5 cjalr $c12, $c17                                  } transfer control flow
6 nop                                               }
7 cmove $c26, $c18                                  } restore $cgp value after call to external_func
```

Listing 4.6: The code from Listing 4.1 compiled for the PLT linkage model.

If we look at the sample C function compiled for the PLT ABI (see Listing 4.6), we can see that it is significantly shorter as there is no longer a prologue that sets up `$cgp`. The omission of the three-instruction prologue can significantly improve performance when calling many short functions within the same DSO. However, the cost of setting up `$cgp` has now shifted from the callee to the caller and therefore performance is not necessarily better — even though the function body contains fewer instructions (see Section 4.8.3). Moreover, the requirement of setting up `$cgp` before invoking a function means that any cross-DSO call must use a trampoline.[17] This also has implications on the uniqueness of function pointers mandated by the C-standard as we cannot use a capability to the DSO-local trampoline (see Section 4.5 for details).

4.2.6 Overview

In this section I have presented the baseline linkage ABIs as well as two fully elaborated pure-capability linkage models with different privilege minimization and performance trade-offs (see Section 4.8). It is important to note that the different pure-capability ABIs (i.e. all but MIPS n64) can be used within the same process. For example, one library

[17] These trampolines must now load two values from memory rather than perform arithmetic on `$pcc`.

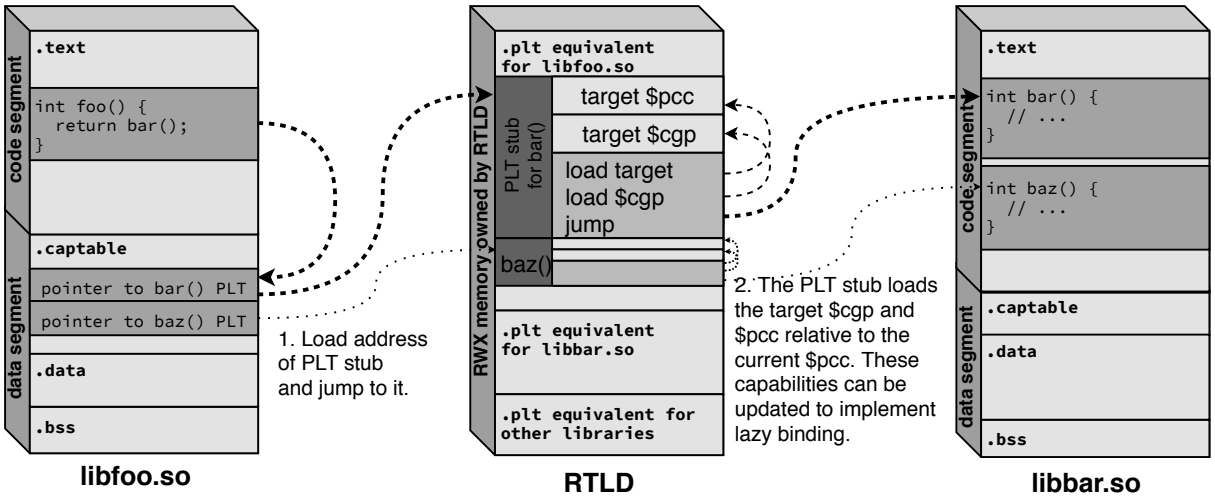


Figure 4.4: CHERI PLT model with PLT located in read-write-execute memory owned by RTLD. Each PLT stub embeds the target code and target data before a short code sequence that loads both and then transfers control to the resolved target function.

could be compiled using the PLT ABI while the main program uses the PC-relative ABI.[18] The ability to interact between different ABIs is provided by the PLT stubs that set up the appropriate `$pcc` and `$cgp` values on library transition. The implementation choices for these PLT stubs are explained in detail in the next section.

4.3 PLT stubs and lazy binding

As mentioned in Section 4.2.5, whenever a transition between libraries is made, it may be required to install a new `$cgp` value. This is performed by the *PLT stubs*, small trampolines that install the actual destination code capability in `$pcc` and load the appropriate `$cgp` for the target library. Traditionally, the PLT stubs are written as part of the DSO by the static linker and the run-time linker is only responsible for filling in relocations and providing the lazy binding resolver that is used by these stubs. For the pure-capability CHERI ABI we chose to deviate from this approach and instead let RTLD dynamically allocate the trampolines.[19] In the current implementation of CHERI pure-capability linkage, PLT stubs consist of two capabilities and a sequence of four instructions (see Figure 4.4).[20]

[18]It is also possible to mix the legacy ABI with the newer replacements if the CheriBSD kernel and RTLD were compiled for the legacy ABI. However, this is not recommended and all support for the legacy ABI will be removed in the future. Nevertheless, this was a useful proof-of-concept, showing the flexibility of my implementation and proving that interaction between the different ABIs is possible.

[19]While this is not strictly the same as a traditional PLT stub (which is part of the caller's DSO), we still refer to these trampolines as PLT stubs.

[20]We could use only one capability and share identical `$cgp` values between different stubs by adjusting the immediate of the load instruction to point to a shared array, but this would significantly complicate the implementation.

4.3.1 Dynamic allocation of PLT stubs

The choice to allocate PLT stubs in RTLD has several advantages:

- Due to embedding the target capabilities in the PLT stubs, we require read-write-execute memory. Having this memory mapped as part of the binary would make this memory accessible to the program. If it is allocated by the run-time linker instead, we can ensure that code outside RTLD only ever receives a capability without write permissions.
- We can use a simple bump-the-pointer allocator and use a shared read-write-execute region for all libraries thanks to fine-grained capability bounds. This avoids wasting memory since we would otherwise have to map multiples of the page size for each DSO.[21] Furthermore, the PLT stub memory must be initialized at run time with the target capabilities and therefore cannot be directly mapped from disk, so we might as well dynamically allocate it.
- Instead of having fixed code sequences in the PLT stubs, we can dynamically choose the PLT contents at run time (see Section 4.3.2).

However, there is one downside to dynamically allocating PLT stubs. Some tools such as the GDB debugger expect PLT stubs to be in the .plt section of the DSO. Breaking this assumption causes GDB to no longer be able to skip over the lazy binding resolver when debugging. Yet, this problem can be addressed by fixing these assumptions in GDB and therefore is not a critical issue. While it may seem that allocating PLT stubs in read-write-execute memory is a problem, only RTLD has full access to the memory and the `captable` contains call-only capabilities (*sentry* capabilities, see Section 4.7.1).

4.3.2 Run-time configurable policy

The approach of dynamically allocating PLT stubs has the advantage that it is possible to use different stub code for cross-library calls depending on run-time policy choices. Instead of loading `$cgp` and directly transferring control to the target function, stubs could also perform more complex tasks. This makes it possible to run an application with differing levels of sandboxing (and the associated performance trade-offs) without recompilation. On other systems this would require a full recompile of not only that application but also of *all* dependent libraries (including libc). Performance-critical code can be compiled and run with the lowest level of compartmentalization whereas high-risk code (such as image processing libraries) can run with the highest level of isolation. One thing that could be done by the more untrusting PLT stubs would be to fix the remaining large privilege leakage flaw in the current ABIs. Currently, the stack is shared between all functions, but a more complex PLT stub could isolate the stack frames between functions (see Section 4.8.4.1 and [68]).

[21] The PLT section would be the only read-write-execute mapping in a binary so cannot be combined with other sections

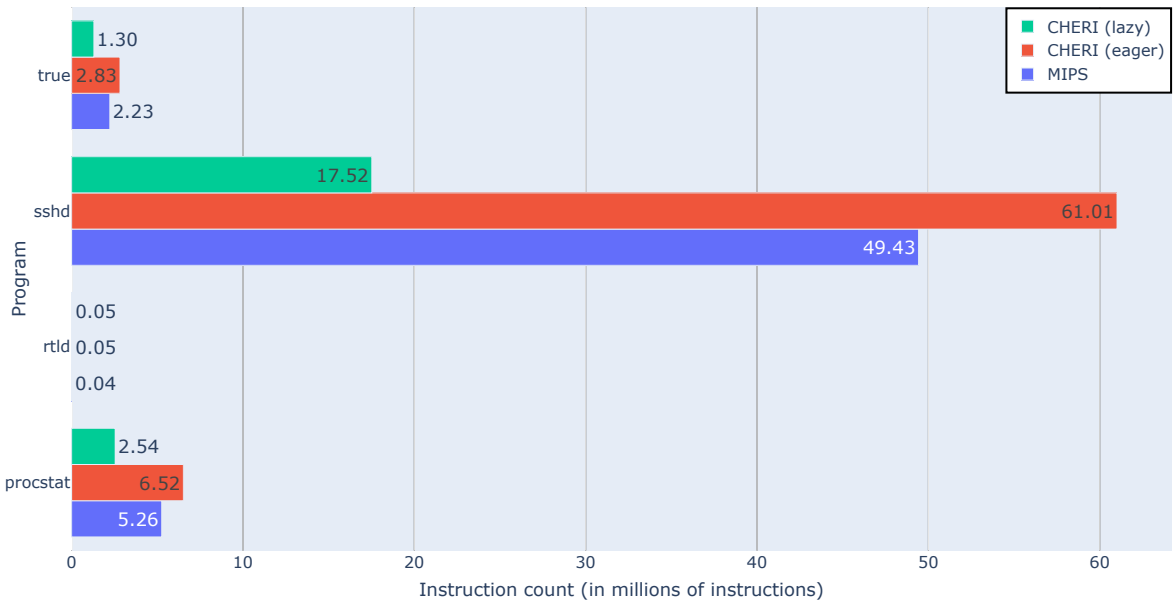


Figure 4.5: Comparison of start-up time (invoking the `--help` option) for various binaries with and without lazy binding enabled. There is no difference for MIPS binaries since the current version of Clang does not generate the code required for lazy binding.

4.3.3 Lazy binding

RTLD must ensure that all `captable` entries have been correctly initialized to point to the target symbols. However, this involves looking up names in a hash table, and is therefore significantly more expensive than writing a fixed value to a given location (as is the case for local symbols). These symbol lookups can be deferred when using PLT stubs, which is referred to as *lazy binding* since the target function is only bound to a concrete address on first use [137, chapter 10]. This can significantly speed up program start-up (see Figure 4.5). For example, `sshd` only needs 17.5 instead of 61 million instructions to print the `--help` message when lazy binding is enabled. However, on most architectures this can come at a cost in run-time performance due to indirection via PLT stubs (see Section 4.8.3).

In FreeBSD MIPS lazy binding is achieved by making all PLT stubs point to an assembly function provided by RTLD, `_rtld_bind_start()`. In the case of MIPS n64, this then calls `_mips_rtld_bind()` with an argument indicating which slot in .got.plt needs to be updated. This function then performs symbol resolution and updates the target pointer. For CHERI, I use a slightly different approach: there is no separate .got.plt section in the DSO that holds the target locations. Instead, the target address and `$cgp` are embedded within the PLT stub and are updated to the resolved values. On program start-up all PLT stubs contain the `$cgp` value for RTLD and point to the CHERI pure-capability version of `_rtld_bind_start()`.

In the PLT ABI, the PLT stubs are always required (as implied by the name), so lazy binding only increases the cost of the first call to an external function. After this first call,

the target `$pcc` and `$cgp` have been written to the PLT stub and the call is as fast as it can currently be.[22] However, in the PC-relative ABI, it would be possible to avoid all the indirection via PLT stubs since the target `$pcc` encodes all the necessary information. In fact, in the PC-relative ABI PLT stubs are only needed for lazy binding, so we completely omit PLT stubs when lazy binding is disabled (e.g. by setting `LD_CHERI_BIND_NOW` or by linking the binary with `-Wl,-z,now`). In this case, RTLD makes the `captable` entries point directly to the target symbol.[23]

4.4 Initializing global capabilities

The CHERI architecture guarantees monotonicity, i.e. it is impossible to create valid capabilities without having access to another one that grants a superset of the rights. Furthermore, capability tags are only stored in RAM, which means that mapping an ELF file cannot include any tag bits. All pointer-type variables in the program — either explicit or compiler-generated (such as C++ *vtables*) — must therefore be dynamically initialized. This is very similar to position-independent executables where the final pointer has to be adjusted by an offset that is not known until run-time. Despite this (minor) overhead of position-independent executables, many Linux distributions are moving towards building all their binaries as position-independent (this is commonly used to improve ASLR [225]).

For CHERI we use the ELF relocation mechanism (see Section 2.3.1) only for external symbol references (e.g. addresses of functions in other DSOs) and rely on a different approach for initializing DSO-local capabilities. The reason for this is that the ELF relocation mechanism is quite difficult to support in statically linked binaries[24] and using different mechanisms for static and dynamic binaries would have added complexity. Furthermore, if we were to reuse the same approach as for external capabilities, we would have to perform symbol table lookups to determine the appropriate permissions for the local capability, which is an unnecessary overhead.[25] Therefore, we decided to use a custom ELF section (`__cap_relocs`) that encodes all the information required for capability initialization.

This relocation mechanism was originally not well suited for dynamic linking and resulted in many inefficiencies. For further details about this initialization mechanism, the problems related to dynamic linking and my approach to solving them, keen readers

[22] It could still be sped up using a new hardware feature: *indirect* `CCall`, as explained in Appendix B.

[23] This could also be done when lazy binding is enabled by overwriting the target function in the `captable` instead of only changing the value in the PLT stub. However, this would require making the `captable` read-write instead of RELRO and has therefore not been done yet. It may seem like a read-write mapping could enable GOT-hijacking attacks [30], but only RTLD has access to the read-write capability and strips write permissions before delegating it.

[24] Originally, we did not support dynamically linked binaries in the pure-capability ABI, so we had to use a mechanism that works for static binaries.

[25] It should be possible to avoid this lookup by storing the additional information in the location where the capability is to be initialized. However, this has not yet been implemented and I propose a slightly different approach in Appendix C.2.4.

may wish to consult Appendix C. While this relocation mechanism is essential for pure-capability linkage, it is not a core contribution of this dissertation and is therefore included as an appendix instead of in the main dissertation body.

4.5 Function pointers

In most contemporary architectures, C language-level function pointers (using the syntax `returntype (*varname)(arguments...)`) are just integer values containing the address of the function.[26] For most other integer pointers, pure-capability CHERI can simply substitute the integer pointer with an appropriately bounded capability. Function pointers are special since they have to encode more information than just the address of the target code: when called, it must also be possible to reach the global data for the current DSO. This is also true for the *implicit* function pointers from the `captable` that are used when calling a function (see Section 4.2). However, C-level function pointers are more complicated than the *implicit* function pointers loaded from the `captable`. For the latter, the *caller's* context (i.e. the current DSO and the matching set of globals) is known. In contrast, a C-level function pointer can be passed across different execution domains such as different DSOs. They can even be passed into the OS kernel when using signal handlers[27] or for starting new threads. As a result, function pointers can be called from *any* context and thus we cannot assume anything about the current `$cgp`. Notably, this is true even for pointers to file-`static` functions.

Invoking a C language function pointer must allow the callee to reach the correct `captable`. As explained in Section 4.2, there are at least three possibilities for this:

- The function pointer is a (*sentry*) capability to a small trampoline that installs the correct `$cgp`. This is the approach taken by the PLT ABI (see Section 4.2.5).
- The globals pointer is derived from another base register that is implicitly set (e.g. the program counter). This allows function pointers to be a (*sentry*) capability pointing directly to the target function without requiring indirection through a PLT stub. This is the approach used in the PC-relative ABI (see Section 4.2.4).
- Function pointers are pointers to a pair of capabilities (a so-called function descriptor). This approach was used by the function-descriptor prototype (see Section 4.2.3).

The following sub-sections explore some complexities that are caused by C-language function pointers.

4.5.1 Guaranteeing unique function-pointer values

The C standard requires function pointers to the same symbol to compare equal [112, §6.5.9.6], and some programs rely on this property. Some architectures do not provide this

[26] Some architectures (e.g. PPC [223], PA-RISC [102] or Itanium [109]) use function descriptors instead.

[27] Interestingly, signal handlers that are registered by `sigaction()` or `signal()` can be invoked either synchronously or asynchronously, so we do not know when the function pointer might be called.

guarantee when the function pointer is cast to a `void*` prior to comparison.[28] However, for CHERI we do want to provide this guarantee when comparing `void*`, even though these casts are an *optional* extension to the C standard [112, Annex J.5.7] The solution I chose to guarantee unique function-pointer comparison is to have the run-time linker resolve all function pointers to a unique trampoline associated with the DSO that contains the function. To distinguish function pointers from function calls during relocation processing, we use two different relocation types: `R_MIPS_CHERI_CAPABILITY` (which must be resolved eagerly) and `R_MIPS_CHERI_CAPABILITY_CALL` (which can point to a PLT stub instead). Essentially, this approach disables lazy binding for function pointers. However, there should be a lot fewer function pointers than direct function calls.[29]

In the PC-relative ABI, we can use a capability that points directly to the target function, so all pointers to the same function are identical. However, in the PLT ABI function pointers must point to a trampoline, which means the run-time linker must ensure they point to the same trampoline. In the current implementation this is done by storing a table of unique trampolines with each DSO. When RTLD looks up a function, it can check in that table if the symbol-defining DSO already contains a unique trampoline and if not allocates a new one and adds it to the table.

Local (file-`static`) and non-preemptible function pointers It is important to note that file-local (`static`) function pointers must also resolve to unique addresses. One more challenge with respect to function pointers in the PLT ABI is the use of non-preemptible (i.e. DSO-local) functions.[30] These non-preemptible symbols will always resolve to a function within the current DSO. In the initial implementation, the static linker emitted a local relocation, which resulted in the function being called without first setting `$cgp`. Surprisingly, this worked in most cases since many of these callbacks did not use global variables or were called from the same DSO. Yet this no longer worked when installing a `static` signal handler since signal handlers are often invoked from another context. The solution to this is to force the creation of a non-local relocation.[31] However, this is not straightforward since local symbols are not preserved in the `.dynsym` section of the binary and therefore RTLD cannot find the corresponding symbol for the relocation. I worked around this limitation by creating a new global symbol with hidden visibility that

[28] In the case of IBM z/OS, function-pointer comparisons may require dereferencing the function descriptor first to compare the address of the target function [106].

[29] Even though C++ *vtables* contain pointers to functions, these entries are different from language-level function pointers. Entries can point to PLT stubs since they are never directly accessible to the programmer and are only used in call sequences.

[30] These can be created by annotating functions with `__attribute__((visibility("hidden")))`.

[31] This is not required when linking statically since all functions share the same `$cgp`.

points to the same function as the local symbol.[32] The hidden visibility ensures that it is not available to symbol lookups from other DSOs.

After making this change, it was possible to boot CheriBSD with only one program crashing in the process. This program was /usr/bin/find and was crashing because this workaround broke function-pointer uniqueness (see Section 4.5.1). The program was comparing a function pointer to a known marker function. However, the LLD workaround added a new symbol every time the address of a local function was taken instead of only adding one new symbol for every static function. Therefore, the run-time linker ended up allocating a new unique trampoline for each function pointer (since the underlying symbol table entry was different even though it referred to the same function).

Function-pointer uniqueness is a rarely relied-upon guarantee, so if it breaks it can result in subtle bugs. The case in /usr/bin/find was caused by the use of a placeholder function pointer to `f_openparen()` (a function that just calls `abort()`) as a marker for an opening parenthesis in the expression tree. When the callback function pointer compares equal to that marker function, /usr/bin/find calls a different function rather than invoking the pointer. Most other application programming interfaces (APIs) that use 'magic' function-pointer values (e.g. `signal()`/`sigaction()`) use integer constants cast to a pointer type instead. Since integer constants cast to function pointers work correctly in the PLT ABI, these issues were not spotted anywhere else. To make this case even less likely to find, the issue only happened if one of the function pointers was declared as a global variable and the other one created inside a function.

4.5.2 Function pointers within RTLD in the PLT ABI

A final challenge with function pointers is the implementation of RTLD itself. RTLD initialization is special since it is responsible for allocating all PLT trampolines but must also relocate itself in order to run. This can cause issues during start-up: to relocate the `R_MIPS_CHERI_CAPABILITY` relocations used for function pointers, RTLD must call other functions such as `malloc()`, etc. However, this can only be done after the `captable` has been initialized. Furthermore, some calls can involve function pointers, thus causing a cyclic dependency.

One solution to this problem would be to defer function-pointer initialization to a second pass over the relocation section and initialize them once all other capabilities inside the `captable` have been initialized. However, it was found that RTLD only needs very few function pointers and the majority of these are created because RTLD links against a static library that contains all of libc built as position independent code. In fact, most

[32] According to the ELF standard it should be acceptable to include local symbols in the dynamic symbol table (if the local symbols are ordered before the global symbols). However, changing the type of the symbol had adverse effects on other parts of the LLD static linker, so the workaround of adding an aliasing symbol was chosen instead.

this code does not need to be linked into RTLD.[33] To address this problem, I added a new flag to the static linker (`-Wl,--warn-if-file-linked=<glob>`) which will emit a warning whenever a `.o` file that matches the pattern is pulled in from a static archive and is linked against. Using this new linker flag (and by making all linker warnings an error), I was able to eliminate almost all the function pointers inside of RTLD. The following six function pointers remain in RTLD: a callback that is passed to `qsort()`, two functions passed to `atexit()`, two callbacks inside RTLD and finally the lazy binding resolver stub. For the cases other than the pointers passed to `atexit()`, we know that they will always be invoked from within RTLD. As a result of this the `$cgp` value will already be correct, and we do not need the trampoline that sets up `$cgp`.[34] For the local pointers, I added a small inline assembly helper, `make_rtld_local_function_pointer()`, that generates a capability to the function without a trampoline.[35] For the two function pointers passed to `atexit()`, I use this assembly helper and then allocate a PLT trampoline manually instead of relying on the C address-of operator.

Besides allowing RTLD to be compiled for the PLT ABI, a side effect of this change was that it reduces the size of RTLD. We no longer pull in unused code from libc, which removes many relocations that need to be processed at start-up. This reduced RTLD's `__cap_relocs` section to 42% of the original size and therefore noticeably sped up RTLD start-up. I have committed this change to FreeBSD where it has reduced the total size of the x86 RTLD by 22% and reduced the relocations within RTLD from 368 to 187. The size changes were even more noticeable in CheriBSD since we were pulling in a lot more from libc due to accidentally added dependencies. The size of the MIPS RTLD was reduced from 361 to 202KiB and the CHERI pure-capability RTLD went from 565 to 375KiB.

4.6 Global Visibility Enforcement

In all linkage implementations listed earlier, the static linker will allocate one capability table for every shared object file. This implies that *any* function in the DSO can access all global variables and functions that are used somewhere in the DSO. Even file-`static` variables, which are only accessible within the current translation unit according to the language model [112, §6.2.2p3], are in fact reachable from anywhere in the DSO. This problem is highlighted in Figure 4.6: the function `foo()` can access all variables in the `captable` (including a variable `local_secret1` that should be inaccessible) even though the only global it needs to access is a function pointer to `bar()`.

[33] For example, all pthread function calls and many system calls are indirected via a function-pointer table. This table exists so that different functions can be invoked depending on whether the current program uses threads or not.

[34] This would no longer be true if we were to compile RTLD with Global Visibility Enforcement (see Section 4.6). However, this would require further changes and is currently not supported.

[35] This assembly helper uses `%capcall20` instead of `%captab20` to avoid the creation of a `R_MIPS_CHERI_CAPABILITY` relocation since we do not require a unique pointer value.

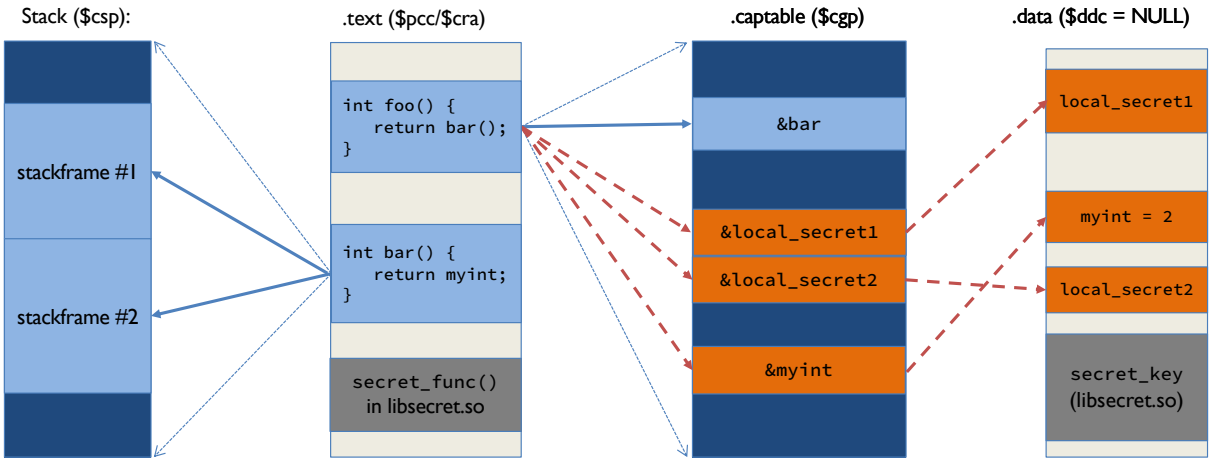


Figure 4.6: The PLT ABI still grants unnecessary privilege. Variables and functions that are not required to be accessible according to the *principle of least privilege* can still be reached through the `captable` (highlighted in orange).

In order to follow the *principle of least privilege* [194, 195] we should ensure that such unnecessary privilege is not available. Even though other entries in the `captable` are not used by the compiler-generated code, they could still be accessed by a malicious attacker who has obtained code injection or has otherwise been called without sandboxing. Therefore, I created a mechanism for least-privilege global variable access using CHERI linkage, which I hereafter refer to as *Global Visibility Enforcement (GVE)*. Similar to how CFI restricts the available control-flow options at any given point in execution, GVE restricts which global pointers[36] can be accessed at any given moment.

In a conventional architecture, global access restriction would have to be implemented using MMU-based compartmentalization or using a software-based implementation relying on compiler instrumentation. The former would be very expensive due to the added translation lookaside buffer (TLB) pressure caused by domain transitions (i.e. one transition for every function call in the finest-grained model). The latter would not be sound, since attacker-injected code would not adhere to these constraints: any integer can be used as a pointer and could therefore be used to bypass GVE unless it is restricted by the MMU.

By building our system upon CHERI architectural capabilities, we can leverage hardware-enforced compartmentalization. The (formally verified [167]) properties of CHERI make it easier to reason about the available privilege: it is limited to the current register file contents and all transitively reachable capabilities. Moreover, all globals are accessed using the `captable` via the `$cgp` register. For the remainder of this section we assume that the current `$cgp` value will not be shared between different contexts.[37] To restrict transitively reachable memory via `$cgp`, we can instruct the linker to emit a

[36]This notion of global pointers includes global variables as well as functions since both are accessed using language-invisible (global) pointers.

[37]In the current implementation it is possible for one function to access the `$cgp` value of a different function, if `$cgp` was saved to the stack. However, this problem could be solved by using a separate, bounded stack for register spills or by limiting the stack (see Section 4.8.4.1).

`captable` for every input `.o` file, thereby restricting code from each file to the symbols used in its source and, as a side effect, securing `static` globals from other object files. Alternatively, we can allocate a table for every *function* to ensure that only the global variables and functions listed and used in the source code[38] can be accessed at run time. Finally, even in the PC-relative CHERI linkage mode we still enforce a coarse-grained form of GVE: it is not possible to access globals outside the current DSO that are not referenced at static link time.

4.6.1 GVE implementation

Currently, the per-function/per-file `captable` model is only compatible with the PLT ABI.[39] The reason for this is that the PC-relative ABI requires a `$pcc` value that spans the text and whole `captable` section (and has permission to load capabilities). Therefore, limiting the scope of only `$cgp` does not provide any advantage as `$pcc` already grants a superset of the access permissions. Furthermore, limiting the scope of `$cgp` is not compatible with the current PC-relative code-generation.[40] In the PLT ABI `$cgp` is a live-in register, and therefore it is possible to set up `$cgp` to point to only the necessary `captable` subset before the call. This can be done by collecting the `captable` subset that is used in each function (or file) at static link time and emitting metadata for the dynamic linker. The dynamic linker can then use this metadata to set up PLT stubs such that each function only gets access to the appropriate subset of globals.

In the static linker globals are added to the `captable` whenever they are referenced by a `R_MIPS_CHERI_CAPTAB*` relocation (or `R_MIPS_CHERI_CAPCALL*` for function calls). This relocation instructs the static linker to add the referenced symbol to the `captable` and replace the relocation with the resulting index into the `captable`. The difference to per-DSO `captable` is that LLD has been modified to track the origin of the `captable` relocation and build separate per-file/per-function tables with their own indices. The final `captable` section in the ELF file is simply a concatenation of the individual tables. Some functions use the same set of globals, so in this case we can merge the `captable` subsets with identical content, to avoid wasting space and increasing initialization time. A more sophisticated implementation could also attempt to find overlapping subsets and partially merge those, but I leave this as future work since this is purely an optimization.

[38] At higher optimization levels this may not correspond directly to the function as written in the source code. Called functions may have been inlined into the current function and therefore, all globals used by those functions will also be added to the set of accessible globals. If this behaviour is not wanted, we suggest compiling with `-fno-inline` to disables inlining while still retaining other optimizations. We could also disable inlining by default when per-function GVE is requested, however, we believe that the performance impact outweighs the theoretical security gain.

[39] It would also work using the function-descriptor model, but that implementation is based on an outdated version of LLVM and is not compatible with the current CheriBSD code.

[40] However, it could be made to work with a slight variation of the PC-relative ABI that has a local function entry point (see Section 4.9).

```
struct CaptableMappingEntry {
  uint64_t FuncStart; // virtual address relative to DSO base address
  uint64_t FuncEnd;  // virtual address relative to DSO  base address
  uint32_t CapTableOffset; // offset into captable (in bytes)
  uint32_t SubTableSize; // sub-table size (in bytes)
}
```

Listing 4.7: Structure used for the GVE `captable` mapping array. An array of this structure is emitted as a .captable_mapping ELF section for processing by RTLD.

Emitting the individual tables is not sufficient to compute the run-time `$cgp` value as it is not yet known which of these `captable` values should be used for a given function. Therefore, I also emit a custom ELF section containing the `captable` mapping data. This section is a sorted array of structures with an offset into the whole `captable` section and the size of the current subset for a given address range (see Listing 4.7). LLD must emit one structure entry for every function that is referenced via a `captable` entry. Additionally, mapping entries must be present for every exported function (even if it is not called from within the DSO) as it could be called from another library or function and RTLD needs to be able to set up a valid PLT stub.

Finally, the static linker adds a dynamic tag `DT_CHERI_CAPTABLE_MAPPING` that points to the `.captable_mapping` section so that the run-time linker can easily find the mapping metadata without walking the section table (which may have been removed using the strip utility). At run time, RTLD finds the `captable` mapping section using the dynamic tag and uses it whenever it needs to perform symbol resolution. When resolving function symbols, RTLD performs a binary search over the `captable` mapping array (since it is sorted by function address) and then creates a PLT stub that loads the `captable` subset as specified in the mapping. This step is sufficient to support per-file GVE. However, when enforcing GVE at a per-function granularity we must also change the `$cgp` when calling file-`static` functions. File-`static` functions do not require symbol resolution, so RTLD must additionally walk the `__cap_relocs` (see Section 4.4) and replace all direct function pointers with trampolines that load the correct `$cgp`. These two steps ensure that every function call is indirected via a PLT stub that loads the correct `$cgp` value.

Enabling GVE Per-file enforcement of global variable accesses can be enabled by passing `-Wl,-captable-scope=file` to the linker. In this case it is also possible to reuse object files compiled for the PLT ABI. However, for *per-function* enforcement (`-Wl,-captable-scope=function`), we must generate slightly different code in the compiler and therefore require this choice to be made at compile time rather than at link time. This is caused by an optimization for the PLT ABI that avoids saving `$cgp` when calling file-local functions, but the assumption that `$cgp` remains unchanged no longer holds when per-function GVE is enabled.

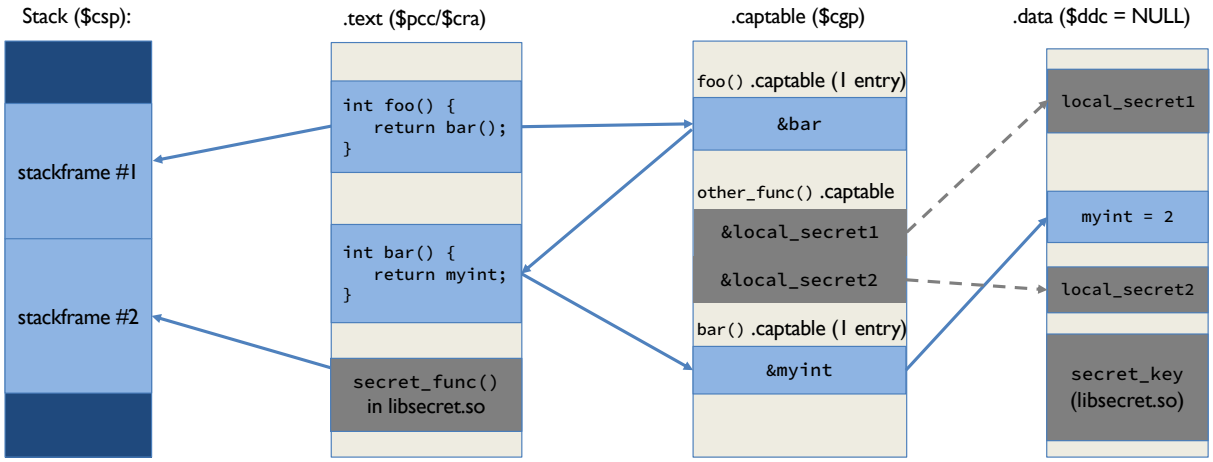


Figure 4.7: Using per-function capability tables prevents variables that are not used in the source program from being accessible via `$cgp`.

4.6.2 Ensuring correct `$cgp` values on function entry

If an attacker were able to call a function with the wrong `$cgp` value, then per-function `captable` would make it easier to load the wrong target function or target global symbol. Per-function `captable` facilitates this because all global symbols will be loaded at a small offset from the `captable` pointer (starting at zero). Therefore, it would be easier to collect a set of functions that load from the correct offsets of an attack-constructed `captable` pointer. This would be less likely to succeed in cases where the function loads from a large constant offset from `$cgp`. However, all pure-capability linkage models include a mechanism to prevent this potential code-reuse attack vector. In the PC-relative ABI, `$cgp` will be derived from `$pcc` and therefore cannot be forged as the location of the code uniquely identifies the matching data. In the PLT ABI, I ensure that programs only ever obtain a sealed capability (an immutable capability that can only be used by jump instructions) to another function and calling this sealed reference ensures that the correct `$cgp` value is loaded.[41] The mechanism of these sealed references will be explained in detail in Section 4.7.1.

4.6.3 GVE security benefit

We argue that by compiling with a globals table per function, the bounds of all code and data pointers can be reduced to the absolute minimum required. This approach has been visualized in Figure 4.7. However, using a per-function table also requires every function call to switch the current `$cgp` and therefore adds at least one additional load to every function call sequence. Furthermore, it is not guaranteed that such tight bounds provide significant reduction in attack surface. Therefore, the performance trade-off must

[41] This is only true in the per-function `captable` case. When using a per-file or per-DSO `captable` it is still the caller's responsibility to provide the correct `$cgp` value for file-local/DSO-local calls. However, in these cases functions will not be using the same indices into `$cgp`.

be evaluated carefully. Our understanding is that limiting the bounds of `$cgp` will almost certainly make it harder to chain calls since the total number of reachable capabilities is smaller. However, CHERI already defends against almost all commodity spatial attacks, and no existing body of ready exploits quantifies this effect.

While the per-library `captable` scope does not follow the *principle of least privilege*, we believe it is a reasonable trade-off between security and performance and is therefore currently the default CHERI linkage model. However, per-file or per-function granularity GVE is provided as an optional feature in CHERI pure-capability compilation and is fully supported. We may reconsider this choice of defaults if we are able to prove practical security benefits of per-function `captable`.

4.7 ISA changes for pure-capability linkage

In order to support efficient and secure CHERI pure-capability linkage I introduced a major new ISA feature, *sentry* capabilities, as well as two minor changes.

4.7.1 *Sentry* capabilities

In the previously mentioned models one significant flaw still exists: a callee can always access the caller's `$pcc` value (using the `$cra` return register) and can load arbitrary capabilities from it. If we are using the default, PC-relative ABI, then this means the callee can also access the whole `captable` of the caller. Therefore, the callee is also able to load any variable and even call non-exported internal functions using the caller's `captable`. Similarly, `captable` entries for external functions could be used to obtain a reference to the `captable` of that DSO.

In my original linkage design, I used `CCall` [170] to transfer control between DSOs to avoid this issue. In this design, every PLT stub must hold a sealed code and data capability for the target library. This requires a unique *otype* value for each library. In the current 128-bit CHERI implementation there are 18 bits of available *otype* space [246, 254]. This would be sufficient as the number of libraries is generally very small. However, if we were to enforce per-file/per-function `captable` we would even require one *otype* for **every function** (or indirection through a trusted switcher) to ensure that each function is called with the correct `$cgp` value. This clearly does not scale to the limited 18-bit *otype*, so I produced a different approach.

To enforce separation between shared libraries, I introduce the architectural concept of *sealed entry capabilities* (hereafter referred to as *sentry* capabilities). These capabilities act in similar ways to regular sealed capabilities; however, they can be invoked without a matching data capability or unsealing permission. Thus, they are an immutable code capability with the only permitted operation being control flow transfer. For this reason, these *sentry* capabilities are ideally suited to implement compartments that do not require

global state (i.e. all data is passed as arguments) or can obtain this global state by deriving it from `$pcc`. After implementing this model, I discovered that *sentry* capabilities are quite similar to the immutable *enter pointers* as proposed by the M-Machine [32]. However, *sentry* capabilities can grant any set of CHERI permissions (including write permission) once jumped to, whereas M-Machine *enter pointers* can only grant a fixed set of permissions.

For the architectural implementation, we reserve one capability object type (currently `-2`) to indicate that a capability is a *sentry*. If this is the case, the only instructions that can be used on this capability without trapping are `CBuildCap` (effectively the equivalent of `CUnseal` if one holds a superset of the *sentry* capability), `CMove`, loads and stores to/from memory (but only as the data operand, not the address operand) and `CJR`/`CJALR` which jumps to the target address, and places an unsealed version of the *sentry* capability in `$pcc`. Moreover, `CJALR` places a *sentry* capability in the link register when invoking a *sentry* capability. This ensures that a callee can no longer read or write via the caller's return address and is only able to return using `CJR`.

The run-time linker can use this feature to seal all external function references in the `captable` as *sentry* capabilities. This restricts the set of readable/writable capabilities especially in the PC-relative ABI, where the `captable` is reachable via the current `$pcc`. If the `captable` contains readable `$pcc` values for other shared libraries, this means that the `captable` for that library is also reachable (and so are all writable data objects). By changing all function `captable` values to be *sentry* capabilities, it is architecturally enforced that the only operation that can be performed is a jump, which makes the callee's `$cgp` value inaccessible.

Currently, *sentry* capabilities are supported in the QEMU, Sail [14] and FPGA implementations of CHERI-MIPS, have been added as an experimental feature to the ISA specification [246, §D.12] and will feature more prominently in the next version.

4.7.2 New load instructions

In the MIPS ISA, the 64-bit load (`ld`) instruction has a 16-bit immediate range. In contrast to this, the original CHERI capability load (`CLC`) only has an 11-bit scaled immediate field. Even though the value of the CHERI immediate is multiplied by 16, this only allows accessing 2^{11} capabilities (and only 2^{10} for 256-bit CHERI). This causes problems for linkage since that immediate field is used to index into the `captable` when loading global variables or pointers to functions. If the `captable` becomes too large to use the immediate field, we have to add two additional instructions to generate a 32-bit index for every global load. This happens very frequently when linking statically[42] as all entries are in the same table. But even with dynamic linkage there are many libraries/programs that

[42]Initially we supported only static linkage, so this caused significant differences in all benchmarks comparing MIPS and CHERI pure-capability code.

require more than 2^{10} `captable` entries. For example, even libc.so contains a `captable` with 5379 capabilities, i.e. over 2^{12}. While it would be possible to use the three-instruction sequence only for non-performance critical functions in libc.so, making this choice would be a manual process that takes a lot of time. Moreover, due to implementation constraints, choosing whether to emit the longer sequence is a compile-time decision. This choice is made with the `-mxcaptable` flag (analogous to the MIPS `-mxgot` flag).[43]

I therefore introduced `CLC` (Big Immediate), `CLCBI`, which reuses a MIPS major opcode (`JALX`) that is not supported by our FPGA MIPS implementation. This new instruction has a 16-bit immediate field which is multiplied by 16 and therefore can be used to access up to 2^{16} capabilities (2^{15} for 256-bit CHERI). This larger range allows us to build most statically linked programs[44] using a single `CLCBI` instruction to access globals. The introduction of `CLCBI` noticeably improved performance (especially for statically linked binaries) since it enabled us to use a single instruction to load a global address instead of using a sequence of three instructions. This was especially noticeable on the CHERI FPGA since it is an in-order CPU and is quite sensitive to instruction-bloat and instruction-cache usage. Moreover, it also reduced the code size of many binaries by over 10% [54].

4.7.3 Thread-local storage

Originally, thread-local storage (TLS) read the MIPS hardware register `$29` and used this virtual address as an offset into `$ddc`. To support bounded thread-local storage, I added a new special-purpose capability register (see Section 3.8.1) to the ISA and Jessica Clarke made the necessary changes to the compiler and operating system to use this register instead the unbounded MIPS virtual address. In doing so, we discovered an issue with capability bounds precision (see Section 3.4): the TLS relocations were adding a constant offset from the TLS block to allow using the entire (signed) immediate range of the instruction encoding. However, this meant that the TLS register had to be 0x7000 out of bounds, which is not representable for programs that do not use many TLS variables. We therefore fixed the relocations to use a zero offset for CHERI pure-capability code. After this change was made, we were finally able to run pure-capability programs with a `NULL $ddc` register, and no longer have unnecessary ambient privilege. However, bounds are currently per DSO and not per variable, to avoid an extra indirection. Furthermore, the TLS register is accessible to all DSOs and values could be loaded from it. A solution to this is proposed in Section 4.8.4.2.

[43] Architectures such as RISC-V use linker relaxations to remove the longer sequence when possible.

[44] Notable exceptions are very large binaries such as DumpRenderTree from QtWebkit which has 120134 `captable` entries and would therefore require a multi-GOT approach when statically linked

4.8 Evaluation

I evaluate CHERI linkage across various dimensions, including security properties and performance (see Section 6.2 for more results for the PC-relative ABI).

4.8.1 Privilege reduction

Prior work on the function-descriptor ABI has shown that fine-grained bounds on `$pcc` can massively reduce the number of large capabilities required to run existing code [42]. The PLT ABI provides the same bounds-minimization guarantees as this function-descriptor prototype. In the case of the PC-relative ABI bounds are not quite as minimal since `$pcc` must span the entire text segment in addition to the `captable`. However, bounds on data are still minimal, so we believe this is a good performance vs. security trade-off. While data and code bounds were minimal in the function-descriptor ABI, it was still possible to reach all other data by loading via the pointers embedded in the `captable`. In this work I have added the guarantee that no additional data is reachable via code pointers since they are now all implemented as *sentry* capabilities. Other than data leakage via other stack frames (which could be addressed using various schemes), there is no way of obtaining a capability that can be used to access non-exported data from a different DSO.

4.8.2 Compatibility

We are able to boot and run CheriBSD using the three supported linkage models (legacy, PLT, and PC-relative). Moreover, we also run the full FreeBSD test suite (in the PC-relative and PLT ABIs) regularly on a continuous integration server and can therefore find regressions whenever changes to RTLD or LLD are made.

Regarding code changes required, the only userspace code that needed adjusting to support different linkage models was RTLD, some inline assembly macros in libc, and a few functions in libc such as `setjmp()` and `dl_iterate_phdrs()`.

4.8.3 Performance

A more thorough performance evaluation comparing CHERI (using only the PC-relative ABI) and the MIPS baseline can be seen in Section 6.2. In this section I only highlight differences between the various pure-capability ABIs.

`qsort()` microbenchmark The impact of linkage ABI can be seen most when calling very short functions. Therefore, I chose the C library function `qsort()` as the worst-case benchmark to run. This function sorts an array and to do so, invokes a callback function for each pair of elements that it is comparing. In this micro-benchmark [187], the comparator is a simple function that compares the value of two integers either ascending or descending

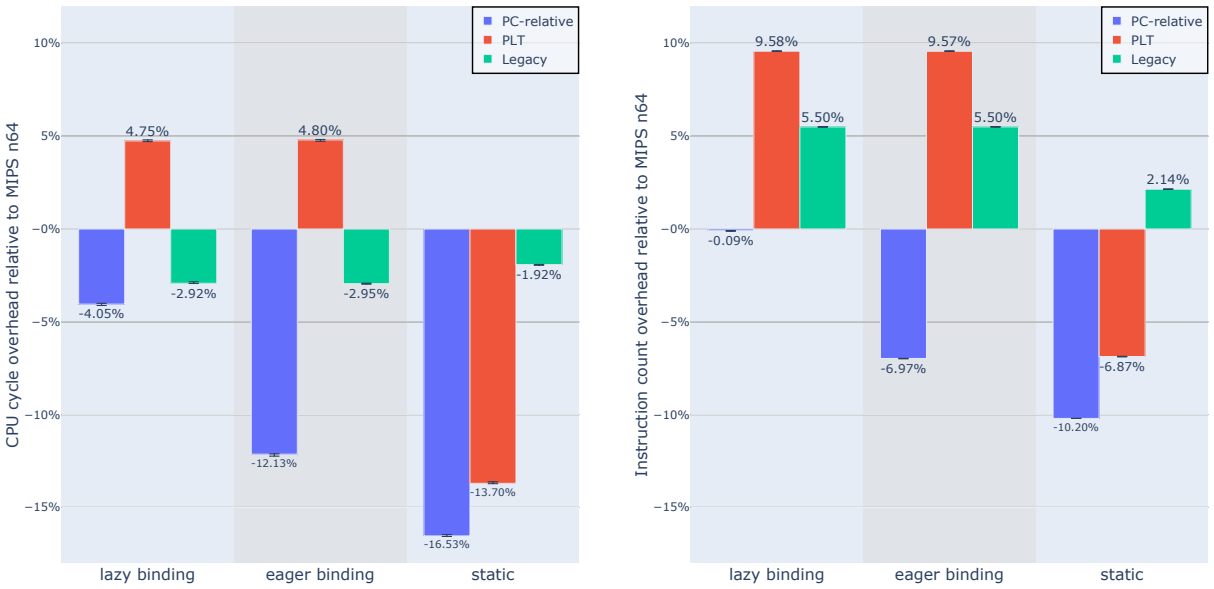


Figure 4.8: `qsort()` micro-benchmark comparing PC-relative, PLT and legacy ABI performance to MIPS n64. We cannot generate MIPS binaries with lazy binding, so in this case we compare with eager binding MIPS n64.

depending on the value of a global variable. To increase the domain transition overhead the actual comparison of the integers is performed in a different DSO. This is actually a realistic use-case: it is common to sort structures based on one string member and therefore call `strcmp()` in the `qsort()` comparator callback.[45] However, to highlight the domain transition overhead my comparator calls a simpler function that simply subtracts two arguments from each other.

Figure 4.8 shows that the PC-relative ABI is fastest for this benchmark. The benchmark clearly highlights the cost of the trampolines needed for the PLT ABI: they add almost 10% instructions and over 8% cycles compared to the PC-relative ABI. The graphs also show that the performance difference is highest in the eager binding case, since the PC-relative ABI can omit PLT stubs and jump directly to the target function. While the legacy ABI has fewer L2-cache misses (not shown in figure) due to the use of a GOT with 64-bit virtual addresses instead of a 128-bit `captable`, the additional instructions needed to create capabilities from the virtual address have much larger impact than the slightly lower number of L2-cache misses.

In this micro-benchmark CHERI is up to 16.5% faster than MIPS. This is caused by a code-generation deficiency in LLVM. For CHERI, the compiler can make use of the split register file and store intermediate values in the additional callee-save registers that are provided by CHERI. Due to fewer callee-save registers, the MIPS version of the benchmark ends up re-loading values from the stack before every comparator call. Even though these values will be in the L1-cache, it still adds multiple cycles overhead in the main benchmark loop.

[45] One of the MiBench benchmarks sorts an array with a `qsort()` callback that uses `strcmp()`.

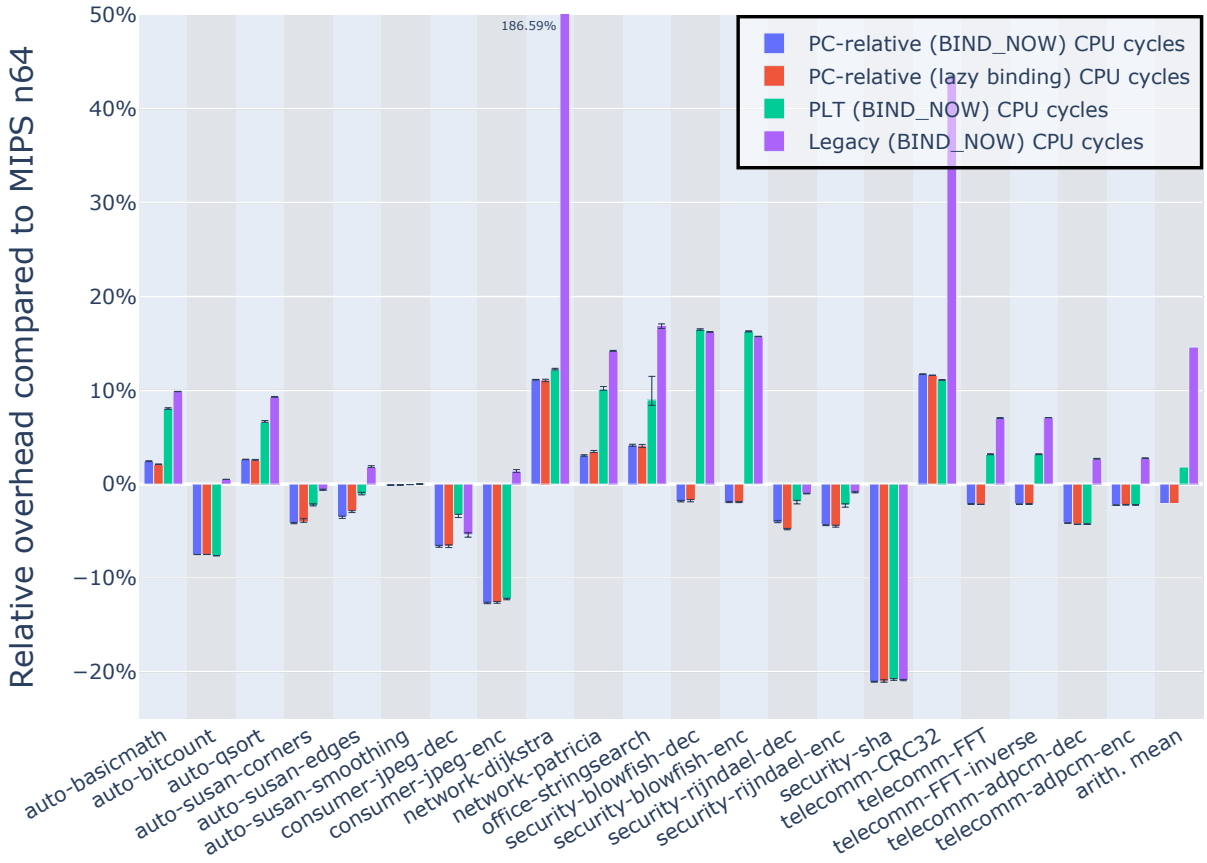


Figure 4.9: MiBench benchmark comparing PC-relative, PLT and legacy ABI performance to the MIPS n64 baseline.

MiBench In the MiBench benchmark suite all pure-capability CHERI linkage models have performance that is comparable to the MIPS baseline (see Section 6.2 for more detailed MiBench performance analysis) and the PC-relative ABI is again the fastest variant. We do not include the lazy binding case for the PLT or legacy ABI since symbol resolution on first call makes no noticeable difference for these benchmarks.

4.8.4 Isolation between shared libraries

The CHERI architecture makes it extremely easy to reason about available privilege. Due to the formally verified monotonicity and integrity guarantees [167], available privilege is limited to memory transitively reachable from the current register file contents. Figure 4.10 highlights how combining per-function `captable` (see Section 4.6) with the new architectural feature of *sentry* capabilities (see Section 4.7.1) reduces the accessible memory to almost the minimum that is required in order to execute. Moreover, it is not possible to obtain capabilities with execute permissions that are not *sentry* capabilities — other than the current program counter (which is bounded to the current function) — as the run-time

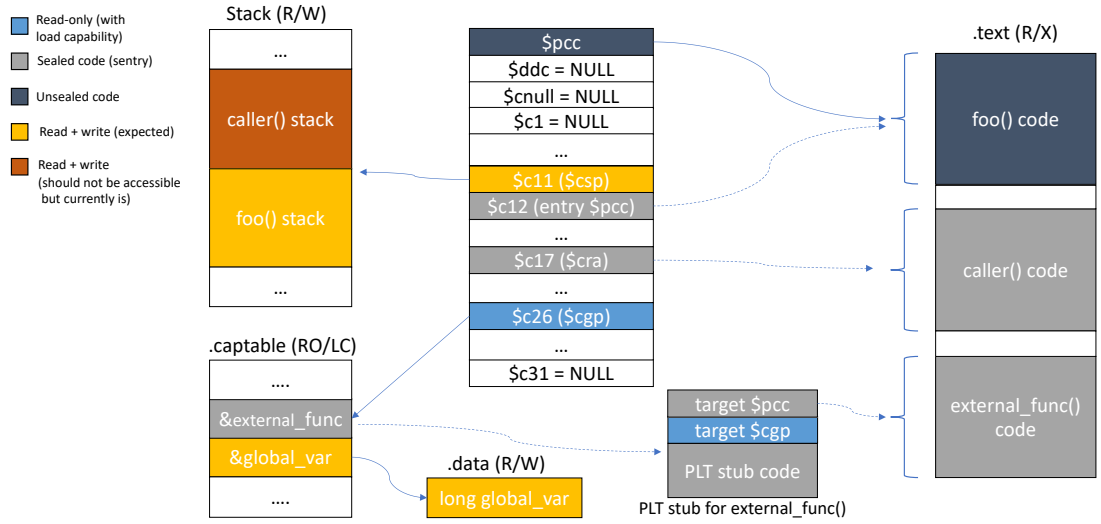


Figure 4.10: Accessible memory inside the sample function (see Listing 4.1) when using per-function `captable` and *sentry* capabilities.

linker ensures that all code pointers in the `captable` are immutable *sentry* capabilities.[46] While traditional ROP attacks are already near-impossible on CHERI, this additionally ensures that control flow cannot be corrupted arbitrarily, only allowing jumps to valid entry points.

However, by default the stack and thread-local variables are shared between individual functions (and even DSOs), so we cannot call this true compartmentalization. The caller's stack frame is still accessible to the callee and therefore all transitively reachable capabilities could be loaded. Writable capabilities to data may have been spilled by a function earlier up in the stack, which a CHERI-aware attacker would attempt to exploit. Once this leakage is fixed, RTLD could be used to implement real sandboxing between libraries, e.g. to completely isolate the memory allocator from all other libraries.

4.8.4.1 Stack isolation

This capability leakage could be avoided by switching to a new bounded stack for each function transition. Figure 4.11 shows how a more complicated PLT stub could provide full isolation between the individual call frames.

At least two options exist in the current CheriBSD linkage design to implement stack switching on domain transition. It is possible to allocate a trampoline on every call, which would result in 11 instructions for stack switching plus the instructions required to allocate a trampoline. Moreover, it requires the callee not to return directly to the caller but instead to a trampoline (via a *sentry* capability to prevent leaking the saved stack) that can restore the previous stack. Such a return path requires four instructions plus trampoline deallocation.

[46] This is not true if a user program such as a JIT compiler uses `mmap()` to allocate a R/W/X mapping. However, CHERI adds the ability to retain capabilities for this mapping only in the TCB and give everyone else read-only capabilities.

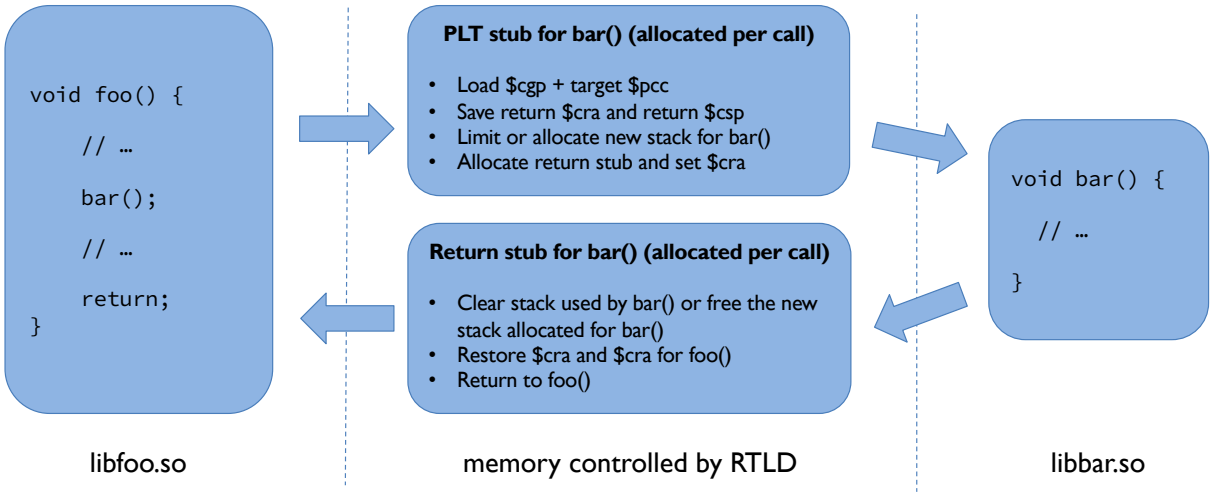


Figure 4.11: Further privilege reduction can be achieved with more complex trampolines.

Alternatively, we could use a reserved software-defined *otype* per shared library, where each library has access to a capability granting sealing permission. This can be used to seal the current stack pointer and return address. To return and unseal the data, a call to a trusted switcher that has the ability to unseal must be performed, thus incurring the cost of a function call on every return. Furthermore, using the limited *otype* space is problematic.

Neither of these solutions is ideal and with the current ISA, stack isolation would require at least 25 instructions for every function call. Therefore, I propose new hardware features, *indirect* `CCall` and *indirect sentries* to avoid these overheads.[47] However, I did not evaluate this technique, so it is not included in the main dissertation body but in Appendix B instead. A complete implementation of library isolation (using a `CCall`-based calling convention) has been done for the clean-slate CheriOS design. In CheriOS, temporally safe stacks and complex PLT stubs ensure full isolation between libraries at a domain transition cost of approximately 200 cycles (function call and return) [68].

4.8.4.2 Securing thread-local storage

As mentioned earlier, bounds on TLS are currently per-DSO and not per variable. Moreover, the TLS block is accessible to all DSOs as we currently store it in a special hardware register that is always accessible. However, this capability leakage could be fixed using the general-dynamic approach of always calling `__tls_get_addr()` instead [62]. To achieve full separation, `__tls_get_addr()` must no longer take a forgeable integer module identifier

[47] For full isolation, we would also have to avoid leaking stale data across calls, e.g. by clearing the stack after every call. In a naive implementation this can be achieved by using a separate stack for each library and de-allocating it on return. It could also be done more efficiently by sub-setting the current stack and clearing modified regions prior to returning [213].

since this allows libraries to ask for TLS variables in a different module. Instead, we would have to validate the caller based on an unforgeable capability.[48]

4.8.5 Implicit CFI provided by CHERI linkage

Another benefit of the CHERI linkage model once combined with the use of *sentry* capabilities is that it provides a form of CFI without any additional performance overhead. The only jump targets that are available to an attacker are those within the bounds of the current `$pcc` as well as any executable capability that is reachable from the current register file. However, the linker ensures that all function pointers and executable `captable` entries are *sentry* capabilities that are immutable and only grant access to the designated entry point. This ensures that only valid jump targets end up in the potential control-flow graph (CFG) available to an attacker: it is not possible to manipulate those capabilities to jump to the middle of a function or instruction as is often done in ROP attacks. Moreover, if we are using the PLT ABI, the current `$pcc` will only grant access to instructions within the current function which is unlikely to contain enough useful gadgets to launch an attack.[49]

New hardware features in upcoming chips provide instructions to partially enforce CFI. Intel includes this as part of their Control-flow Enforcement Technology (CET) [108] extensions whereas ARM v8.5a includes a feature called Branch Target Indicators (BTI) [86]. Both of these CPU extensions provide landing pad instructions (which are no-ops in older versions of the ISA to ensure backwards compatibility). When enabled, every indirect jump must target one of these landing pad instructions or otherwise the CPU issues a trap. If these landing pads are placed at the beginning of functions and/or jump table targets, the set of possible indirect jumps can be restricted to only valid jump targets. This type of CFI is a weak level of protection since any valid jump target is reachable and not just the jump targets that should be reachable from the current function.

The CFI properties provided by CHERI linkage and *sentry* capabilities are stronger than landing pads for cross-function jumps. Firstly, only jump targets in the current DSO's `captable` as well as function pointers or C++ *vtable* entries in objects reachable from the current register file are available. This set will almost always be a strict subset of all valid jump targets. Secondly, we can also use a per-file `captable`, in which case only functions that should be called from the current context are reachable.[50] However, we do not provide protection for jump-table targets within the current function (which could be a lot of targets in a JavaScript interpreter main loop switch). In this case it could

[48] CheriOS uses a different approach to solve the TLS problem: all functions receive a thread-local pointer instead of `$cgp` and then load `$cgp` from that thread-local table [68]. A similar approach, using a table of tightly bounded thread-locals, could also be used in CheriBSD.

[49] This is already assuming the attacker has found a gadget to copy valid code capabilities and execute them later (which is a lot less likely and more difficult on CHERI than on traditional architectures). The integrity and monotonicity guarantees of CHERI capabilities ensure that there is no way of fabricating a code pointer that is not a subset of those currently available.

[50] Function pointers and C++ *vtables* still add to this set of reachable targets. However, we can assume that they were correctly initialized and are *sentry* capabilities pointing to the beginning of a function.

```
foo: # external entry point
    lui $1, %pcrel_hi(_CHERI_CAPABILITY_TABLE_-8)
    daddiu $1, $1, %pcrel_lo(_CHERI_CAPABILITY_TABLE_-4)
    cgetpccincoffset $cgp, $1
    # Set bounds on $pcc and  $cgp. This might need additional instructions if bounds
    # do not fit in the CSetBounds immediate range
    csetbounds $cgp, $cgp, __cap_table_end - __cap_table_start
    # Set bounds on $pcc and jump to .L.foo.local
    cincoffset $c12, $c12, .Lfoo.local - foo
    csetbounds $c12, $c12, .Lfoo.end - .Lfoo.local
    cjr $c12
    nop  # delay slot (can be filled with the csetbounds on $cgp)
.Lfoo.local:  # entry point for function calls from the same DSO
    # actual function body
.Lfoo.end:
```

Listing 4.8: Assembly code for a function foo with an external and a local entry.

still be beneficial to use something like the BTI/CET landing pads to prevent jumps to unintended targets. Finally, returns are only possible to valid code capabilities that the current context already has access to and therefore CHERI linkage also provides a limited form of backward-edge CFI.

4.9 Future work

In this chapter I have presented the current CHERI pure-capability linkage models and shown that they can be used to run an entire operating system, CheriBSD, while providing basic isolation between shared libraries. Nevertheless, some aspects of the linkage models could still be improved. This section will list some potential future changes.

A hybrid between the PLT and PC-relative ABI With the addition of *sentry* capabilities to the architecture it is possible to create a linkage model that provides a tightly bounded `$pcc` and `$cgp` value, yet still derives the `$cgp` value from `$pcc`. This can be achieved using a *local* and an *external* entry point for functions similar to the approach taken by the PPC ELFv2 ABI [140].[51] The external entry points to a short code sequence that computes `$cgp` from `$pcc` in the same way as the PC-relative ABI does. It then adds two `CSetBounds` instructions that bound `$pcc` and `$cgp` appropriately. As all the values used in this are known at static link time, it also means this does not involve any text relocations. Listing 4.8 shows the code for such an external entry prologue.

This approach is similar to inlining a PLT stub before the real function entry and omitting it for local calls (similar to not using a PLT stub for calls within the same library). However, it has one advantage over the PLT ABI: the external function prologue does not involve any memory loads, consists of only 7 instructions (or up to 11 if neither the `captable` nor the current function can be bounded using the immediate operand version of `CSetBounds`) and it is located immediately before the actual function target. This

[51] To guarantee function-pointer uniqueness, function pointers must always use the external entry point even when called from within the same shared library.

should reduce the data-cache footprint and should fetch the initial instructions of the target function into the instruction cache, thereby avoiding the potential cache misses and delays caused by a PLT stub. According to IBM, this dual-entry approach results in better performance than function descriptors/PLT stubs [140]. Moreover, removing indirect jumps is beneficial for the relatively simple CHERI-MIPS branch predictor.

All entries in the `captable` will be *sentry* capabilities that point to either the local entry point (for direct function calls from within the same DSO) or the external entry point (for function pointers and calls to functions in other DSOs). As these *sentry* capabilities cannot be dereferenced or modified, the full-DSO capability that is used to derive the `$cgp` and `$pcc` value is only accessible for a few instructions between the external and the local function entry. It would also be possible to make this model work with per-file and per-function `captable`. However, this would require (almost) every function call to use the external entry.

Defending against speculative execution leakage Using an explicit notion of compartments in the hardware that prevents micro-architectural speculation from leaking values [121, 141] across compartment boundaries could furthermore eliminate speculation-based information leakage attacks [249]. This is especially important considering that current software-based mitigation techniques such as x86 *retpolines* come at a high cost as they essentially disable branch prediction or other forms of speculation.[52] We could extend trampolines between security-critical DSOs to add a `CSetCID` instruction to change the micro-architectural notion of the current compartment [246, §7.4]. This prevents some speculative execution vulnerabilities if the processor does not to share micro-architectural state between compartments.

4.10 Conclusion

In this chapter I have presented the opportunities and challenges in supporting dynamic linking on a CHERI pure-capability system. As typical of CHERI work, this effort has seen much co-design between software and hardware. One of the core contributions is the addition of new ISA features such as untyped sealed capabilities (*sentry* capabilities) which can be used for compartmentalization without using up the limited 18-bit object-type space that is provided architecturally. I have also presented a PC-relative linkage model that provides basic isolation between individual DSOs while incurring almost no performance overhead compared to MIPS and increased performance compared to the insecure baseline model. If one wants stronger isolation guarantees, this is also possible with higher overhead in the current scheme. Finally, I have shown that the CHERI linkage model can be used to enforce the new concept of Global Visibility Enforcement (GVE, see Section 4.6). GVE

[52] For the `qsort()` benchmark from Section 4.8.3, I measured a 10x slowdown with GCC and 4x with Clang when enabling *retpoline*.

also guarantees coarse-grained CFI by default and fine-grained CFI (albeit without the return-edge protection) when using a per-function `captable`. Moreover, this can be done without the overheads that are often incurred by other CFI schemes [28].

5 ENFORCING SUB-OBJECT BOUNDS

Prior research on spatial memory protection has mostly focused on detecting and preventing accesses beyond the current C object (or more commonly, the underlying memory allocation for that object). This approach is popular because it avoids complexity but offers less granular bounds checking (and therefore provides incomplete spatial safety). This was also the initial approach taken for assigning bounds in pure-capability CHERI programs [53, 54]. However, spatial memory safety techniques can only be considered *complete* if they also prevent invalid sub-object accesses [209].

In this chapter I present a novel technique, *CHERI sub-object hardening (CheriSH)*, that — building on top of pure-capability C/C++ (see Chapter 3) — can enforce memory protection at a much finer granularity. Using this technique, we are also able to protect against out-of-bounds accesses at the *sub-object* level. CheriSH therefore completes the memory-safety guarantees provided by pure-capability C/C++ by enforcing intra-object spatial safety in addition to the existing inter-object safety. This is the first time CHERI has been used to guarantee **complete spatial safety** [209] of unsafe languages such as C and C++. Whilst the existing pure-capability CHERI protection model already prevents most exploits, sub-object bounds further reduce available privilege and thus limit the impact of any potential attack.

This chapter is structured as follows: After introducing the concept of sub-object spatial protection in Section 5.1, Section 5.2 presents the guarantees provided by CheriSH as well as the design choices that guide the implementation. Next, I describe the underlying conceptual model as well as the implementation based on LLVM in Section 5.3. Section 5.4 evaluates CheriSH across the following dimensions: source-level and run-time compatibility with existing C/C++ code, the performance impact compared to a pure-capability C/C++ baseline, improvements to memory protection (based on buffer-overflow test suites as well as real-world issues discovered by CheriSH) and finally the implementation complexity. I present potential future changes to CheriSH in Section 5.5, contrast the difference between CheriSH and related work in Section 5.6 and finally conclude this chapter in Section 5.7.

5.1 Introduction

Enforcing bounds at allocation granularity is not always sufficient to prevent buffer-overflow attacks. For example, the 2008 Linux kernel `vmsplice()` root exploit [44] was the result of a sub-object overflow [228]. This exploit obtained arbitrary code execution by overflowing a nested array of `struct page` in such a way that the kernel invokes a user-controlled

`struct page` destructor function pointer. Additionally, in 2018 Gil, Okhravi and Shrobe published a paper proving that the spatial safety protection provided by many defence techniques can be bypassed using only intra-object overflows [82] (which they refer to as a *Pointer Stretching* attack). They also presented a real-world attack that works against NGINX even when the recent *low-fat pointer* schemes for protecting heap [64] and stack [65] memory are enabled. This attack might also work in the presence of AddressSanitizer (ASan) since it does not enforce sub-object protection [174, 253]. Figure 5.1 shows an example of a structure that could be used for such an attack: writes to the `name` field allow corruption of the `fnptr` member if sub-object bounds are not enforced.

On contemporary architectures, sub-object overflows are primarily used for pointer injection, but in a post-CHERI world (where referential safety prevents pointer injection), they could be exploited for data-oriented attacks against other fields — e.g. to cause improper state-machine transitions, violate invariants, etc. Although pure-capability C/C++ does not protect sub-objects, attackers have a much more limited ability to exploit this weakness since CHERI's referential integrity guarantees and the CFI guarantees provided by *sentry* capabilities (see Chapter 4) make it extremely difficult to launch a code-reuse attack. Nevertheless, narrower bounds are a form of privilege minimization that is important for preventing these kinds of attacks.

To address this shortcoming in pure-capability C/C++, I added support for CHERI-based sub-object protection to Clang and committed the first prototype of CheriSH in June 2018. When pointers are taken to sub-objects, those derived pointers have bounds narrowed to the specific field to which the pointer is taken. This protection technique has since matured, and we are now able to boot all of CheriBSD with intra-object protection — while having made only minimal changes to the source code. In the following sections, I explain the implementation of CheriSH and explore the practical challenges I encountered in developing a sub-object bounds scheme that is viable in large-scale C/C++-language corpora such as an entire operating-system kernel and userspace.

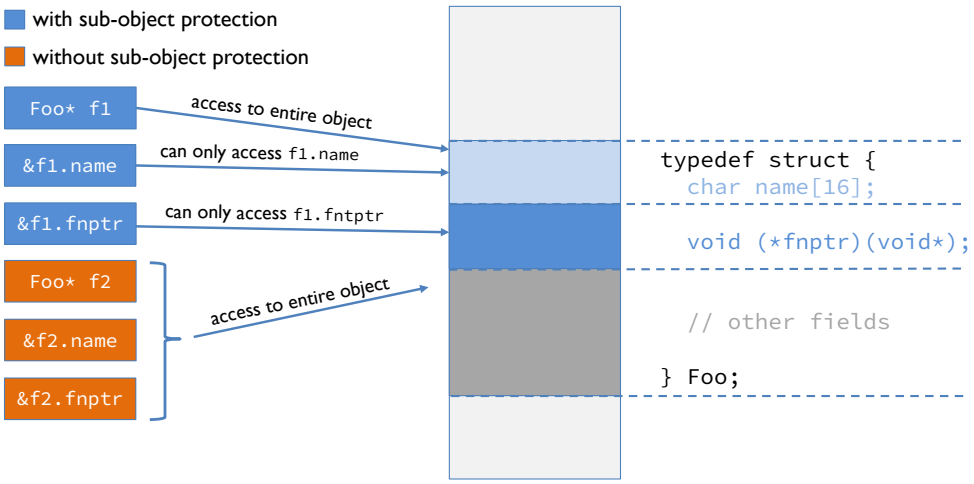


Figure 5.1: Example of a structure containing an array that could be used to overwrite a function pointer if sub-object bounds are not enforced.

5.2 Design principles

The implementation of CheriSH builds upon the strong spatial memory-safety guarantees provided by CheriABI [54], the ABI used for pure-capability C/C++ in the CheriBSD operating system. The remainder of this chapter uses CheriABI to refer to the pure-capability C/C++ baseline without support for sub-object protection. Starting from this baseline we chose the following design goals for our implementation of CheriSH:

No explicit checks on memory accesses Although adding software checks could potentially make the implementation faster and simpler, this has downsides. If we were to rely on software checks prior to loads (as used by e.g. ASan), a single file compiled without the checks breaks the soundness of the spatial memory safety. For CheriSH, a file compiled without enhanced bounds results in reduced safety *in that compilation unit*, but all others will still have the tight bounds. Due to the hardware-enforced bounds and monotonicity of capabilities this also applies to all data passed to the file without tight bounds.

Resilience to untrusted or malicious code Many other memory-safety approaches cannot protect calls to code compiled without the instrumentation. For CheriSH, we want to ensure that untrusted code must adhere to the policy imposed by CheriSH. In fact, we go further and insist that even attacker-injected code must not be able to violate sub-object spatial safety. The monotonicity and integrity guarantees provided by CHERI capabilities ensure that compilation units compiled without CheriSH must also honour the access permissions that the caller intended to provide.

Protection for assembly code Instrumentation-based approaches cannot work for inline assembly or functions written in assembly. Rigger et al. report that 28% of analysed

open-source projects contain assembly code [189]. We want CheriSH to also support these projects and ensure that assembly code cannot break the protection.

Fine-grained control over bounds We are aware that some existing code may not be compatible with sub-object bounds (see Section 5.4.1 for examples) and therefore CheriSH must provide the ability to opt-out of tight bounds for certain incompatible patterns.

Offer compatibility versus protection trade-offs We want to provide a sane default mode that should mostly work without any source code changes (`-cherish=safely`) and a mode in which the compiler will attempt to set bounds whenever possible unless there is an explicit opt-out annotation (`-cherish=aggressively`).

Binary compatibility with non-CheriSH code Legacy codebases might contain constructs that are incompatible with CheriSH (see Section 5.4.1) and replacing them may not be feasible. Therefore, it must be possible to disable CheriSH for individual files (or whole libraries) while retaining binary compatibility with CheriABI. Additionally, the different modes of CheriSH must also be binary compatible and it must be possible to mix them even in the same program or library.

No reliance on interposing system libraries Many other memory-safety tools that provide binary compatibility with uninstrumented code (e.g. ASan, Intel Memory Protection Extensions (MPX) or SoftBoundCETS) rely on intercepting calls to core system libraries such as libc. This is required to model the memory effects and update the bounds metadata. However, this approach is very fragile (see Section 5.4.7) and composes badly with other interposition, e.g. by debugging tools. Therefore, one design principle of CheriSH is to not require interposition.

Fail-closed rather than fail-open policy Instead of omitting or nullifying bounds (as happens with Intel MPX), i.e. a fail-open policy, we want to retain all sub-object bounds even when passing pointers to uninstrumented code. We consider CheriSH a *security feature* and not a debugging tool. Therefore, we believe it is better to have a fail-closed policy than to retain compatibility with existing code by omitting bounds protection.

Resilience to compiler bugs Compilers are extremely complicated programs and therefore almost certainly contain bugs that in turn lead to bugs in generated code. By building upon CHERI hardware and not only the compiler, we can rely on monotonic, hardware-enforced bounds that ensure prevent compiler bugs from amplifying available access rights. Importantly, this allows the implementation of CheriSH to be as simple as possible and do nothing more than monotonically reduce the bounds for a given sub-object whenever this is made possible by the source language.

```
1  void use_reference(long&);
2  typedef struct {
3    long value;
4    // Accesses to this buffer must not allow modification of the function pointer!
5    int buffer[10];
6    fn_ptr_t callback;
7  } struct_with_array;
8
9  void example(struct_with_array *s, long index, long nbytes) {
10   // (1) Taking a pointer to the member x will bound to sizeof(long):
11   memset(&s->value, 0, nbytes);
12   // (2) The array decay from int[10] -> int* sets bounds to 10*sizeof(int), i.e. 40 bytes:
13   memset(s->buffer, 0, nbytes);
14   // (3) Accessing the array at a non-constant index adds bounds:
15   s->buffer[index] = 10;
16   // (4) Reference binding will also narrow the bounds to sizeof(long):
17   use_reference(s->value);
18 }
```

Listing 5.1: Example code highlighting the four cases where bounds can be narrowed.

5.3 Model and implementation

For CheriSH, we follow the *principle of least privilege* and take the view that any reference (i.e. a pointer or a C++ reference) to a sub-object should be bounded to the size of the sub-object. By narrowing bounds for sub-objects and by building upon monotonicity, spatial and referential safety provided by CHERI, CheriSH can provide complete spatial safety. In the case of C programs, we identify three cases in which we can further tighten bounds for sub-objects. For C++ programs, we must handle these three cases as well as C++ references (since they are also implemented as CHERI capabilities). These four cases are highlighted in Listing 5.1 and are explained in the following paragraphs.

Address-of operator The most obvious occurrence of sub-object bounds (and initially the only supported case) happens when a programmer passes a pointer to a structure member to another function. For example, in line 11 in Listing 5.1 there is the potential for a sub-object overflow if `nbytes` is larger than the size of the field `s->value`.[1] Whenever the compiler is invoked with CheriSH enabled, it will bound most address-of operator results to the size of the object whose address is being taken. For example, in the expression `&foo.a` the result will be a capability with the base pointing to `foo.a` and the size being `sizeof(foo.a)`.[2] We can use the size of the expression for almost all address-of operators, but there are some cases related to arrays where this is not true. In general, this address-of operator bounding also applies to C++, except that we do not (redundantly) narrow bounds when converting a C++ reference to a pointer. These special cases are discussed in Section 5.3.3.

[1] This overflow can actually be the intended behaviour in some cases (see Section 5.4.1.3).

[2] We also do this for to the `__builtin_addressof()` compiler built-in that behaves in the same way as the address-of operator but — unlike the operator — cannot be overloaded in C++.

Array decay Another case where sub-object bounds can prevent intra-object overflows is so-called *array-to-pointer decay*. C allows passing an array to any function that takes a pointer and when doing so implicitly converts (i.e. decays) the array to a plain pointer. However, this loses the original size information, and the caller must assume that the callee will not read or write outside the bounds of the array. In CheriABI, we already provide a limited form of protection against this kind of overflow: the pointer passed to the called function will only grant access to the original allocation that contains the array. For global arrays or plain arrays allocated on the heap or stack, this allocation will be identical to the bounds of the array.[3] This is sufficient to prevent access to another object but if the array is embedded inside another structure, it does not prevent erroneous (or worse, malicious) code from writing beyond the bounds of the array and into other members of the structure. Therefore, with CheriSH every fixed-size array-to-pointer decay will have bounds set to the size of the array.[4]

Array subscripts Originally CheriSH set bounds only on address-of and reference operations. However, we discovered that some sub-object bounds overflows are plain array buffer overflows caused by incorrect uses of the array subscript operator on fixed-size arrays.[5] Whenever we encounter code such as `s->buffer[n]`, CheriSH inserts an appropriately sized bounds-setting instruction (see Section 5.3.3). This case is similar to the existing UndefinedBehaviorSanitizer (UBSan) `-fsantize=array-bounds` instrumentation. However, `-fsantize=array-bounds` does not perform the bounds check if the size of the array is not known at compile time (e.g. for flexible array members) whereas for CheriSH this check is implied by prior bounding and enforced by the hardware. CheriSH ensures that both accesses before the flexible array member (see Section 5.3.3) as well as accesses past the end of the containing allocation are caught.

C++ reference binding Finally, in C++ we can narrow bounds whenever a reference is taken. Unlike C pointers — which can either point to a single element or an entire array — C++ references can always be narrowed since language standards specify that they always refer to a single value [111, §11.6.3]. In LLVM, C++ references and pointers are implemented using the same intermediate representation (IR) type. Therefore, we can use the same compiler intrinsic to tighten the bounds on C++ references and pointers. This bounding can be performed both for implicit creation of references (e.g. the `use_reference(s->value)` expression in Listing 5.1) and for explicit creation of a reference from a pointer using the dereferencing operator. In the former case we can always narrow the bounds, whereas the latter case may not always be safe (see Section 5.3.3).

[3]Modulo padding due to alignment and compressed capability precision (see Section 3.4).

[4]However, there are special cases caused by programmers declaring variable-size structures as fixed-size arrays (see Sections 5.3.3 and 5.4.1.2).

[5]For example, all BOdiagsuite tests (see Section 5.4.5) that were not caught by pure-capability compilation were caught by this instrumentation.

5.3.1 Clang implementation

The implementation of CheriSH builds upon the CHERI Clang compiler. Whenever we encounter a pattern where we could narrow the bounds of a pointer (e.g. taking the address of a structure member), the compiler frontend inserts a `llvm.cheri.cap.bounds.set` intrinsic. The backend lowers this intrinsic to a `CSetBounds` instruction, which ensures that the bounds are correctly narrowed at run time. Most importantly, I was able to make all required changes in the C/C++ frontend without having to modify the backend. The desired protection properties are already provided by the existing CHERI LLVM backend, so we only need to insert finer-grained bounds (see Section 5.4.6)

5.3.2 Opting out of sub-object bounds

There are cases where CheriSH is not compatible with existing C and C++ source (see Section 5.4.1 for details). In these cases we provide two options. It is possible to lower the level of protection provided by CheriSH (or completely disable it) on a file-by-file basis. However, source files might only contain one or two incompatible expressions. Therefore, we also provide fine-grained annotations to disable bounds narrowing only for certain expressions or types.

5.3.2.1 Per-file choice between protection and compatibility

For cases where existing code is not immediately compatible with CheriSH, we provide the ability to choose between multiple levels of strictness using compiler flags. Users of CheriSH currently have the choice between six (progressively stricter) levels.

In the *conservative* mode all bounds are set to the entire object. This is currently the default compilation mode. The *references-only* mode additionally sets bounds for C++ references but does not attempt to bound address-of expressions. In *subobject-safe* mode we bound all pointers to the sub-object level. However, we use the full structure bounds when taking pointers to union members. Additionally, pointers to arrays will use the bounds of the entire array rather than bounding one element. The following three modes are more aggressive but not always compatible with existing C code. In the *aggressive* mode, we add tight bounds for unions and assume that `&array[not_constant]` expressions refer to a single array element rather than the full array. The *very-aggressive* setting is the same as *aggressive* except that address-of expressions using constant array indices are now bounded to a single element. Finally, in *everywhere-unsafe* mode we set bounds whenever possible (i.e. unless there is an explicit opt-out).

To achieve minimal bounds, it would be best to always use the most aggressive setting. However, we acknowledge that this is not a good trade-off between compatibility and security since it breaks too many existing programs. In general, we recommend using the *subobject-safe* mode as this should be compatible with most code (see Section 5.4.1)

and still provide strong security guarantees. Even though pointers to arrays will retain the bounds of the entire array rather than those of an individual element, this mode still prevents any kind of overflow between different structure members, thus providing complete spatial safety.

5.3.2.2 Fine-grained control over sub-object bounds

In addition to per-file configuration, CheriSH also adds opt-out annotations that instruct the compiler not to tighten bounds for a specific expression. We currently provide the following approaches to make previously incompatible code work with CheriSH:

Completely disable sub-object bounds It is possible to annotate a `typedef`, record member, or variable declaration with `__attribute__((cheri_no_subobject_bounds))` to indicate that the compiler should not tighten bounds when taking the address or creating a C++ reference. When compiling in C++11 (or C2x [113]) mode this attribute can also be spelled as `[[cheri::no_subobject_bounds]]`. Since adding this attribute would result in a warning when compiling with a compiler that does not support CheriSH,[6] we added a `__no_subobject_bounds` macro to `sys/cdefs.h` that expands to nothing if the compiler does not support CheriSH.

Disable sub-object bounds in specific expressions We also provide a new compiler built-in, `__builtin_no_change_bounds()`, which can be used to opt-out of automatically tightening bounds for a given expression. Due to implementation constraints, this built-in must wrap the expression that would have bounds added and not the operation that causes the bounds to be added. For example, `&__builtin_no_change_bounds(x->a)` will retain the bounds of `x` when taking the address of the member `a`, but the similar expression `__builtin_no_change_bounds(&x->a)` will use the size of member `a` to set tight bounds. It can also be used when indexing arrays and for array-to-pointer decay: `__builtin_no_change_bounds(foo->array)[10]` will use the bounds of the surrounding `foo` object instead of the declared size of `array`. This built-in is needed e.g. to implement a FreeBSD macro, `__PAST_END(array, offset)`, that is used to access a variable-size array that has been declared with a fixed length.[7] As this spelling is rather verbose, the system-provided header `sys/cdefs.h` includes a `__unbounded_addressof(expr)` macro (see Listing 5.2) that uses the built-in if available and expands to `&expr` for compilers that do not support CheriSH.

Explicitly specifying sub-object bounds size It is also possible to opt-out of automatic bounds narrowing and explicitly set them using `__builtin_cheri_bounds_set()`.

[6] This happens during the CheriBSD build since certain binaries in the source tree are bootstrapped with the host compiler.

[7] This macro was required prior to C99 since flexible array members did not exist. Even though C99 was standardized 20 years ago, five files in the FreeBSD source tree still use this macro.

```c
// The following macros are defined in <sys/cdefs.h>
#define __unbounded_addressof(expr) (&__builtin_no_change_bounds(expr))
#define __bounded_addressof(expr, size) \
  ((typeof(&(expr)))__builtin_cheri_bounds_set(__unbounded_addressof(expr), size))

char *foo(struct str *strp) {
  // request &strp->str_array bounded to 12 bytes
  return (__bounded_addressof(strp->str_array, 12);
}
char *bar(struct str *strp) {
  // Inherit the bounds of strp for &strp->str_array
  return __unbounded_addressof(strp->str_array);
}
```

Listing 5.2: Definition and sample usage of new macros for customizing sub-object bounds.

```c
struct message {
  int m_type; /* and possibly other members ... */
  /*
   * Variable-length message data: pointers taken to this sub-object will have a
   * lower bound at the first address of the array, but inherit the upper bound
   * from the allocation containing the array, rather than always 252 bytes higher.
   */
  char m_data[252] __attribute__((cheri_subobject_bounds_use_remaining_size(252)));
};
```

Listing 5.3: Example use-case for the remaining-size attribute.

However, doing so is unnecessarily verbose and requires casts from `void*` which can be error prone. Therefore, I added `__bounded_addressof(expr, bytes)` for bounding sub-objects with custom sizes (see Listing 5.2) to the system-provided header sys/cdefs.h. So far we have only needed this macro for one case: treating adjacent structure members as a contiguous array (see Section 5.4.1.3).

Use remaining allocation size In certain cases, the size of the sub-object is not known, but we still know that data before the structure member will not be accessed. Pre-C99 code will often declare such members as fixed-size arrays (see Sections 5.3.3 and 5.4.1.2) which will cause a hardware exception on `CSetBounds` if the allocation does not grant access to that many bytes.[8] To use the remaining allocation size instead of completely disabling bounds (and thus protecting against buffer underflows), the annotation `__attribute__((cheri_subobject_bounds_use_remaining_size))` can be used.[9] The annotation has an optional argument to specify an upper bound in bytes for cases where this is known. An example showing how to use this attribute can be seen in Listing 5.3. By adding the `cheri_subobject_bounds_use_remaining_size` attribute to the field declaration, we can prevent buffer underflows that overwrite previous members (such as the message type). While using the remaining size still allows overflowing into consecutive objects (if all objects are part of the same allocation, as would be the case with `struct dirent` and the `getdents` system call), we do prevent all underflows. If

[8] If flexible array members are declared using the C99 syntax (with empty square brackets), the compiler will automatically use the remaining allocation size.

[9] The GCC-MPX compiler also provided a similar attribute `bnd_variable_size` for this case [79], but it appears they inherit the entire bounds of the parent structure, thus still allowing underflows.

the actual run-time size of the trailing array is known, bounds that do not overlap with consecutive objects could be set explicitly using macros or compiler built-ins. However, adding these macros increases the amount of code that needs to be adjusted compared to a single annotation of structure member (or no change at all for C99 flexible array members). Additionally, the actual size of the variable-length object may not be known when a pointer to it is taken.

Use different code when CheriSH is enabled As a final fallback, if code is not compatible with sub-object bounds, it is possible to conditionally compile different code using the C pre-processor. To allow this, the compiler will define the macro `__CHERI_SUBOBJECT_BOUNDS__` when CheriSH is enabled.

5.3.3 Special cases

As noted earlier, sometimes we cannot unconditionally narrow bounds, even though we might statically know the size of the object.

Taking address of array elements Taking the address of an array member is ambiguous as we do not know if the programmer intended to create a pointer to an individual element or a pointer to the whole array with an offset (e.g. when passing a pointer to the last element for reverse iteration). To ensure compatibility, the default CheriSH compilation mode treats expressions in the form `&array[index]` as referring to the entire array.[10] In the *aggressive* setting we use the following heuristic: for `func(&array[0])` and `&array[LAST_INDEX]` the full array is used. However, for a constant index to the middle of the array or a non-constant index, we set the bounds to a single element. Originally, we assumed that non-constant indices generally indicate loops where the programmer meant a single element (e.g. `for (i = 0; i < end; i++) { func(&array[i]); }`). However, we found multiple cases across CheriBSD where this was not true[11] and therefore only set tight bounds for non-constant indices in *aggressive* mode.

Variable-size arrays CheriSH does not additionally set bounds on variable-length array types on the stack (e.g. `int stack_array[length]`) [113, §6.7.6.2] because the LLVM backend handles this case already.[12] However, many C programs also contain variable-length structures. Code targeting standards prior to C99 often uses fixed-size arrays of length one (or zero [77]) to work around the absence of flexible array members [113,

[10] We originally tried using a single-element pointer unless annotated, but this caused too many incompatibilities.

[11] /bin/sh uses the same `char` buffer to allocate a series of strings. This code uses the expression `&stringbuf[current_index]` and expects the result to be bounded to multiple elements. Similar code was also discovered in /bin/csh and libiconv.

[12] Clang supports variable-length arrays only on the stack, so (unlike GCC) we do not have to handle the more complicated case of variable length arrays in structures [78] (not to be confused with flexible array members [113, §6.7.2.1.18]).

```
class Foo { /* ... */ };
class Bar : public Foo { /* ... */ };
void do_something(const Foo& value);
void example(std::vector<Foo*>& vec, int* iptr, int& iref) {
  for (Foo* f : vec) {
    // Bounding to sizeof(Foo) might not be safe since do_something()
    // could cast Foo& to Bar& or call virtual functions.
    do_something(*foo);
  }
  Bar b;.
  do_something(b); // Setting bounds to sizeof(Bar) is safe since the static type is known.
  use_int_ref(*iptr); // Setting bounds as type is int and original pointer might have larger bounds.
  use_int_ptr(&iref); // No need to narrow bounds to since the reference will already be bounded.
}
```

Listing 5.4: Example code showing special cases for C++ references.

§6.7.2.1.18] (for which CheriSH can use the remaining allocation size as described in Section 5.3.2.2). This appears to be common (see Section 5.4.1.2), and therefore we default to using the remaining allocation bounds for arrays of size zero or one at the end of structures.[13]

C++ references In C++ it is possible to take the address of a reference to turn it into a pointer. When using CheriSH, this poses a question: should this address-of be bounded based on the referenced type or can we trust the bounds of the reference? Originally, we were setting bounds to the size of the referenced type: using the address-of operator on a `char&` would create a one-byte capability. However, this can cause problems if the dynamic type of the reference is larger than the static type. In this case narrowing the bounds could remove access to the sub-class fields. Furthermore, the only case where this would tighten the bounds on any capabilities would be if the caller were compiled without CheriSH since otherwise all references are already tightly bounded. Additionally, not narrowing bounds improves the code-generation for code that forwards a reference parameter to another function call as we can omit `CSetBounds` when creating a reference from an existing one.

Moreover, we cannot tightly bound references created from pointers to C++ structures since the run-time type could be larger (see Listing 5.4). To avoid problems in this case, CheriSH only narrows bounds if the record is marked as final and does not have a *vtable*.[14] It is important to note that this problem only applies when creating references from pointers to record types, but not for fixed-size types such as `int`. We also bound pointers and references to C++ classes and structures whenever the size is statically known (e.g. when allocated on the stack).

[13] In the *aggressive* mode we set strict bounds here, but *subobject-safe* is designed for maximum compatibility (even with arguably wrong/outdated code).

[14] However, few classes are declared as final and therefore we cannot use this property very often.

5.3.4 Capability precision

The precision of compressed capabilities affects sub-object bounds similarly to inter-object isolation (see Section 3.4). To avoid overflows between distinct objects, I modified the compiler to increase alignment and insert tail-padding for stack, heap and global variable allocations. There are four possible choices to address this for sub-objects:

- We could automatically pad all structures so that the bounds of every member[15] have a size and alignment that can be bounded precisely.
- A less invasive approach would be to add compiler warnings (which could be promoted to errors) that trigger whenever taking sub-object bounds would result in an imprecise capability. The compiler can then suggest the correct `_Alignas()` value that needs to be added to the given field to guarantee representability.
- We could introduce a new attribute — e.g. `ensure_representable_bounds` — that lays out the annotated structure or type in a way that guarantees precise bounds. This is similar to the first option but would allow this automatic padding to be added only for specific types that are considered important from a security point-of-view.
- Finally, we could also not attempt to provide precise bounds and always use the imprecise `CSetBounds` instruction.

Automatically inserting padding to guarantee precise sub-object bounds could result in overly large objects — even if the address of a member is never taken, it might increase the alignment requirement for the entire structure. It is important to note that padding must be added even when compiling without sub-object bounds, as otherwise structure layouts would change depending on this compiler flag and result in ABI-incompatible libraries. Therefore, capability precision becomes part of the ABI if we automatically insert padding. Another problem with the automatic insertion of padding is that it cannot work for types with C99 flexible array members as the size of those is not known until run-time. This effect could be mitigated by padding to ensure that at least a given number of elements in the array is representable. The current standard explicitly allows this behaviour: 'the size of the structure is as if the flexible array member were omitted except that it may have more trailing padding than the omission would imply' [113, §6.7.2.1.18].

Clang already increases the alignment of a structure if a flexible array member has an `_Alignas()` specifier that is greater than the current structure alignment. We could therefore increase the alignment of flexible array members to guarantee a minimum representable size, if we can ensure the padding is the same between compiler versions and invocations. However, this approach could break existing code that assumes that a flexible `char` array will not increase the alignment and size of the structure. Additionally, choosing the size requires guidance by the programmer to avoid excessive padding, and is therefore no better than manually specifying the required alignment.

[15] This includes nested structures which may be large and/or oddly aligned.

The other option, adding warnings suggesting an explicit `_Alignas()`, would also increase the alignment for non-CHERI architectures, so a better solution could be a target-specific attribute that ensures precise capability representability. However, I chose not to implement the attribute since it could differ between compilers (and even compiler versions) and might result in structures not being compatible unless compiled with the same compiler version.[16] Moreover, unknown attributes can be ignored by the compiler,[17] so a typo in the attribute name could silently result in padding not being added.

In the current design, we view sub-object protection as a best-effort feature and therefore use (potentially) imprecise bounds. Nevertheless, bounds should rarely be imprecise as the current 128-bit CHERI compression scheme guarantees precise bounds for any object smaller than 4096 bytes and most sub-objects are below this threshold.

5.3.5 Debugging sub-object bounds

To aid programmers' understanding of when sub-object bounds will be used (and to help debug opt-out annotations), I added the compiler option `-Rcheri-subobject-bounds`. With this flag the compiler will emit a diagnostic remark every time it adds sub-object bounds listing the type that was used and the size of the sub-object bounds. These diagnostics include the size of the bounds, the name of the variable and the reason for the narrowing of bounds: `setting sub-object bounds for field 'values' ( array subscript on 'struct Foo [3]') to 12 bytes`. We also emit diagnostics whenever an operation that would normally add sub-object bounds did not narrow bounds: `not setting bounds for array subscript on 'int * __capability' (array subscript on non-array type)`.

Furthermore, to differentiate sub-object violations from object-granularity violations, I added a new compiler flag `-mllvm -cheri-subobject-bounds-clear-swperm=<NUM>`. When enabled, we will clear the software-defined CHERI permission bit `NUM` whenever the bounds that are inserted by CheriSH are smaller than the existing bounds. Whenever a process crashes CheriBSD prints a register dump that includes the CHERI permission bits and the register that caused the fault. If this register does not have the software-defined permission bit `N`, we know that the error was detected by CheriSH and would not have been caught otherwise. Since this debug feature consumes one of only three available permission bits and adds four additional instructions (two `CGetLen`, one compare and one conditional move) every time bounds are tightened, it is not enabled by default.

Finally, we have discovered that debugging sub-object violations is usually very straightforward. When a bounds fault is triggered by sub-object bounds, we found that the cause

[16] This is not just a theoretical problem: the compression algorithm is non-trivial, and we have had to make various bug-fixes to the C library that computes the required alignment for a given size. Therefore, it is currently true that different versions of the compiler would produce incompatible structure layouts.

[17] Many projects are compiled with low warning levels and might not see the 'attribute ignored' warning.

	container_of	Variable length	Multiple members	Unions (default)	Unions (aggressive)	Inheritance	Reference widening	Other
FreeBSD 3rdparty	3*	3§	0	0	(1)†	12(4)‡	0	0
FreeBSD libraries	2*	1§	0	0	(1)†	3	0	0
FreeBSD headers	1*	3§	0	0	(1)†	0	0	0
FreeBSD programs	0	0§	1	0	(2)†	0	0	1
FreeBSD kernel	≈ 22⊕	4§	4	0	(1)†	≈ 10⊕	0	0
MiBench	0	0	0	0	(0)†	1	0	4
libc++	0	0	0	0	(0)†	0	6**	0

* All changes were located by compiler errors after modifying the `container_of` macro.
** All but one of these changes were suggested by compiler diagnostics.
⊕ We did not compile and test all drivers so there are probably more instances of this pattern.
§ There were many more structures with a trailing size-one array. However, by default CheriSH treats these as variable-size arrays so we did not have to make any changes.
† In the default CheriSH compilation mode this does not cause any issues. The numbers in parentheses are estimates for more aggressive compilation modes (see Section 5.4.1.4).
‡ This could be reduced to four changes by adding a new attribute (see Section 5.5).

Table 5.1: Summary of CheriSH changes. The numbers in each row are the number of files that required modification.

for this violation is usually in the same function or at most one or two stack frames above. When using GDB, we can therefore locate the cause of the error very quickly.

5.4 Evaluation

I evaluate CheriSH across various dimensions such as memory protection benefit, performance overheads, discovered bugs, implementation complexity and most importantly compatibility with existing code.

5.4.1 Compatibility

Contrary to our initial assumptions, making existing C and C++ code compatible with CheriSH requires very few modifications to the source code, compiling and running successfully. However, we did discover compatibility issues while running test suites for FreeBSD and the libc++ test suite. Some C idioms are incompatible with sub-object provenance and therefore require opt-out annotations or code changes. Table 5.1 contains a breakdown of the changes required, and the following sub-sections provide more detail

```c
#define container_of(ptr, type, member) ({ \
    const typeof( ((type *)0)->member ) *__mptr = (ptr); \
    (type *)((char *)__mptr - offsetof(type,member));})

struct generic_list_entry {
    /* next element (generic_list_entry and the not containing struct)*/
    struct generic_list_entry *next, *prev;
};
/* Structures can be added to a linked list by including a list_entry field */
struct mytype {
    int value;
    struct generic_list_entry list;
};

struct mytype* next_element(struct mytype* ptr) {
    /* First obtain a pointer to the generic_list_entry member in the next element */
    struct generic_list_entry* next_entry = ptr->list.next;
    /* Then derive a pointer to the containing mytype structure using container_of() */
    return container_of(next_entry, struct mytype, list);
}
```

Listing 5.5: Sample usage of `container_of()` macro to implement a generic linked list.

on these incompatibilities. The table also shows that, if changes are required, these are almost exclusively located in library code or system headers rather than in program code.[18]

5.4.1.1 Obtaining pointers to the parent structure (`container_of`)

Some C projects rely on the ability to obtain a pointer to the containing structure from a pointer to a nested structure. This can be used to implement 'generic' data structures in C (such as sys/queue.h in FreeBSD). These data structures generally rely on embedding the container metadata (e.g. a linked list `next` or `previous` pointer) inside the structure that is added to the container (see Listing 5.5 for an example). This idiom is common in operating-system kernels: the Linux kernel uses the macro `container_of` [126, 127], FreeBSD also provides a macro with identical functionality using the spelling `__containerof` and the Windows kernel includes a similar `CONTAINING_RECORD` macro [152].

A pattern similar to `container_of` was found in uthash.h, part of a popular C library providing implementations of hash tables and other data structures [98]. In this case, the macro `ELMT_FROM_HH` subtracts an offset (the `hho` member of `UT_hash_table`) from the hash table handle (the address of `struct UT_hash_handle` that is nested in the structure that should be stored in the hash table). This is almost identical to the `container_of()` macro, the only difference being that the offset is a run-time constant instead of a compile-time one.

All uses of `container_of` require that pointers to the nested structure can be converted to a pointer to the containing structure. When compiling with CheriSH, this can be achieved by annotating the embedded structure with the `cheri_no_subobject_bounds` attribute. As noted in Section 5.3.2.2, this attribute ensures that taking the address of the sub-object returns full bounds and that the pointer can be converted to the containing

[18] Table 5.1 does not split the *3rdparty* code into library versus program, but so far all required changes have applied to libraries or headers.

structure without being out-of-bounds. Annotating the nested structure ensures that all uses of the macros work as expected. Most incompatible uses of `container_of` were found with compiler diagnostics: I modified the `container_of` macro definition to check that the structure member is annotated with an opt-out attribute (using a new `__builtin_marked_no_subobject_bounds` built-in), and if not it will generate a compiler error when compiling with CheriSH enabled.

The amount of changes due to this incompatibility is small since adding an attribute to a single structure or structure member fixes it. Overall, I had to annotate six files in the CheriBSD userspace: sys/queue.h, one internal header in libpmc and libthr and three copies of uthash.h.[19] These numbers are consistent with the findings of Chisnall et al. [41]: in the userspace software packages that they evaluated, only `perf` used the `container_of` pattern.[20] In the CheriBSD kernel this pattern is used much more frequently, but so far we only had to add opt-out annotations to 22 files.

5.4.1.2 Variable-size fields declared as fixed-size arrays

I found a total of 6 structures in the FreeBSD source tree that include an array field declared with a fixed size although it has a variable size at run time. The most commonly used structure is `struct dirent` which is used by the libc function `readdir()` to list directory entries. This is a commonly used function — e.g. by `sh`, `ls` and many other core UNIX utilities. `struct dirent` contains a `char d_name[MAXNAMLEN + 1]` member and has a comment stating 'name must be no longer than this' which indicates the reason for using a fixed size array. Moreover, this code was almost certainly written before C99 so there was no way of declaring a flexible array member. At run time the array will usually contain less than `MAXNAMLEN` bytes (the actual length can be determined by reading the `d_namelen`/`d_reclen` members). Furthermore, it will usually be contained in a buffer immediately followed by another `dirent` structure. This pattern causes compatibility issues since taking the address of the `d_name` member with CheriSH enabled will create a capability bounded to `MAXNAMLEN` bytes. The resulting `CSetBounds` operation will trap at run time if the buffer containing the contiguous `dirent` structures has less than `MAXNAMLEN` bytes remaining.

One fix would be to add the opt-out annotation to avoid tight sub-object bounds for this member and instead always use the underlying buffer size. However, we can do slightly better with the `use_remaining_size` annotation (see Section 5.3.2.2), which will result in a capability starting at `d_name` and ending at the end of the buffer. This allows us to prevent any out-of-bounds accesses before the start of `d_name` (which could easily happen in string-processing code). As mentioned in Section 5.3.2.2, the `use_remaining_size`

[19] The FreeBSD source tree contains multiple copies since it bundles third-party software that already includes uthash.h.

[20] Based on the experience in FreeBSD, `perf` could probably be fixed with a single type annotation rather than having to annotate all 156 uses of the `container_of` macro.

annotation takes an optional integer argument indicating the maximum size, which allows us to bound the `d_name` field to at most `MAXNAMLEN` bytes without explicitly adding `CSetBounds` instructions to all uses of `struct dirent`. For minimal bounds, we should load the `d_namelen` field and set bounds based on that value instead.[21] The in-kernel directory entry structures for the UFS, EXT2FS, and FAT filesystem also contained the same fixed-size array and had to be updated.

I also noticed this pattern in another commonly used structure, `fd_set`. The `__fd_mask` structure member is declared as having a length of 128, but programs that know they need fewer file descriptors may allocate less. This issue was found in `sshd` which heap-allocates a `fd_set`, derives the size from a variable `maxfd` and thus allocates less than the full `fd_set`.

The `sockadddr_nb` structure in `netsmb/netbios.h` was also using a fixed-size array instead of a C99 flexible array member. In this case, the size of the array could be up to 64 bytes, but it was declared as having a length of 34 bytes. Additionally, I annotated three copies of the GCC-derived `obstack.h` header that was using a 4-byte `char` array as a variable-size structure. I also saw this pattern in many files in FreeBSD with a one-element array at the end of a structure to indicate variable-size data. As this appears to be common,[22] the *subobject-safe* CheriSH compilation mode will assume that such a member is a variable-size array.

Finally, I saw a related issue in the `libc` time-zone code. When parsing the time-zone file format, a structure for the file header is followed by other structures. However, the code was obtaining a pointer to the trailing data using `&p->last_field+sizeof(p->last_field)`. With CheriSH, this sets the bounds to the size of `last_field` and the address to the first byte past the resulting limit, so the first access to the trailing data will fault.[23] While this is not quite the same pattern as in the other cases, it is also treating a notionally fixed size structure as a variable-size one.

5.4.1.3 Using multiple structure members as a contiguous array

In some (rare) cases, a pointer to a structure member may be passed to another function assuming that the following members can also be accessed. For example, this could be used to initialize or reset multiple members at once with `memset()` or `memcpy()`. This case can be fixed by annotating the member or call-site with an opt-out annotation (which would then yield full allocation or container bounds) or an explicit bounds-setting intrinsic.

[21] Currently, this needs to be done manually at each call-site, but we are considering a new annotation that references a field containing the size and could be used by CheriSH to set the tightest possible bounds. However, we have not yet implemented this and are not sure how often this would be useful.

[22] This is true at least for pre-C99 code such as large parts of FreeBSD.

[23] The fix for this was very simple since the structure was already contained in a union with a `char` buffer of the maximum size. Instead of deriving the buffer pointer from the header structure, we add the size of the structure to the start of the buffer.

```
void assemble_udp_ip_header(unsigned char *buf, int *bufix, u_int32_t from,
    u_int32_t to, unsigned int port, unsigned char *data, int len)
{
    /* initialize struct ip and compute checksum: */
    struct ip ip = ...;
    ip.ip_sum = wrapsum(checksum((unsigned char *)&ip, sizeof(ip), 0));
    /* Initialize udphdr and compute checksum: */
    struct udphdr udp = ...;
    /* Note: Taking the address of an int but expected range is two integers */
    udp.uh_sum = wrapsum(checksum((unsigned char *)&udp, sizeof(udp),
        /* Compute checksum across ip_src and ip_dst by passing a pointer to ip_src */
        checksum(data, len, checksum((unsigned char *)&ip.ip_src, 2 * sizeof(ip.ip_src),
            IPPROTO_UDP + (u_int32_t)ntohs(udp.uh_ulen)))));
}
```

Listing 5.6: Simplified example of /sbin/dhclient using a pointer to one `uint32_t` member (`ip_src`) as a contiguous 8-byte buffer in the innermost call to `checksum()`.

Alternatively, the adjacent members could be wrapped in a nested structure, and a pointer to that structure could be used instead.

One example of this pattern happened during CheriBSD boot when dhclient attempts to obtain a DHCP lease. As can be seen in the simplified example in Listing 5.6, the code computes a checksum for the IP source and destination address by passing a pointer to the `ip_src` member and expects this pointer to also be valid for accesses to the next structure member. /sbin/dhclient contains this pattern twice: once in `decode_udp_ip_header()` and once in `assemble_udp_ip_header()` (see Listing 5.6). I fixed this problem by using the `__bounded_addressof` macro (see Section 5.3.2.2) to bound the pointer correctly. This pattern was more common in the FreeBSD kernel. The process structure contains a `p_startcopy` and `p_startzero` member that is used as a marker for ranges that need to be zeroed or copied when forking a new process. We also found that the compatibility layers that are required to run binaries using different ABIs on the same kernel were copying parts of layout-compatible structures using pointers to members in the middle of the structure. We believe that this pattern could be common in some codebases and therefore programmers using CheriSH should be aware of it.[24]

5.4.1.4 Taking address of union members

Another source of incompatibility is code that expects that a pointer to one `union` member can be used to refer to other members. In the cases where I saw this in CheriBSD, this was caused by limitations in the POSIX socket APIs. For example, the `getnameinfo()` function expects a pointer to a `struct sockaddr`, yet the dynamic type of this structure could be different depending on the network protocol (e.g. `struct sockaddr_in` for IPv4 or `struct sockaddr_in6` for IPv6 addresses). In FreeBSD, a common pattern of dealing with this API appears to be wrapping these different structures in a single `union`

[24]It would be possible to emit a compiler warning when a pointer to a member is passed to `memcpy/memset` with a size parameter greater than the size of the field. We have not yet added this check but are considering it as future work.

```
typedef union sockunion {
        struct sockaddr_storage ss;
        struct sockaddr         sa;
        struct sockaddr_dl      sdl;
        struct sockaddr_in      sin;
        struct sockaddr_in6     sin6;
} sockunion_t;
void example(sockunion_t* p, void* buf) {
  getnameinfo(&p->sa, p->sa.sa_len, buf,  sizeof(buf), NULL, 0, NI_NUMERICHOST);
}
```

Listing 5.7: Simplified example of ifmcstat using a pointer to a union member to refer to the entire structure.

type, `union sockunion` (see Listing 5.7) and passing a pointer to the `struct sockaddr` member.

I first noticed this pattern in /usr/sbin/ifmcstat but have also observed the pattern of using `union sockunion` in other files. Therefore, the *safe* mode of CheriSH currently defaults to using the bounds of the entire union when taking the address of one member. However, this only applies to top-level union members, and we do narrow bounds when taking a pointer to a member of a nested structure embedded in the union.

5.4.1.5 Emulating inheritance in C

I discovered that some libraries were simulating C++ inheritance in C by embedding the 'superclass' structure as the first member in the 'derived' type. This pattern is generally referred to as *physical subtyping* [35, 208] and usually works with CheriSH, yet some libraries perform upcasts using `&ptr->base` instead of `(struct Base*)ptr` to avoid an explicit cast. The latter case works with CheriSH, but in the first case we will return a pointer bounded to the size of the 'superclass' member. Due to monotonicity a following downcast will not widen the bounds and will produce a bounds violation once a 'subclass' member is accessed. While the C standard states that a 'pointer to a structure object, suitably converted, points to its initial member […], and vice versa' [113, §6.7.2.1.15] (i.e. they point to the same address), it does not mandate that these pointers are identical. This implies that the standard does not prohibit narrowing bounds for the first structure member as long as the address remains the same.

In userspace, I have so far encountered this pattern in five libraries and fixed the problem with one annotation per 'subclass'. We therefore believe that this issue can be fixed easily. In libexpat, I had to annotate a total of three structure members; five in libarchive; two in liblzma; three in `fts()` implementations in libc;[25] and finally one more in libelf. This could be reduced to one per 'class hierarchy' in each project by providing a new opt-out annotation for C 'inheritance' that is applied to the 'base class' `struct` (see Section 5.5).

[25] FreeBSD contains three almost identical copies of the same file to maintain backwards compatibility with older binaries.

We saw this pattern more regularly in the kernel and have so far annotated ten files. However, there are many more drivers that we have not yet tested that almost certainly use this idiom. Nevertheless, we believe that this idiom is rare enough that we do not need to accommodate for it by default. If we did, bounds would not be tightened in the more common case where the first member is not used for 'inheritance' or `container_of`.

GCC's MPX implementation defaulted to using the containing structure's bounds when taking the address of the first member. It is likely they made this choice to improve compatibility with code that takes the address of the first member for inheritance.[26] However, this design choice leads to them missing some RIPE [252] buffer-overflow tests [174]. Moreover, avoiding narrow bounds for the first member seems to be rarely necessary, so for CheriSH we default to always narrowing bounds and require opt-out-annotations if this pattern is being used.

5.4.1.6 C++ reference widening

All the issues listed above for C also apply to C++, however, they are generally less common in C++ codebases. So far, I have discovered only one C++-specific incompatibility: attempts to widen the bounds of a reference. C++ references should refer to a single object, but I found cases where programmers expected a reference to a single array element to grant access to the entire array after using the address-of operator. Multiple instances of this issue were also found in libc++, a C++ standard library implementation. For CheriSH-compatibility, I had to change only six files, and all but one of the incompatibilities could be identified using compiler diagnostics.

Taking the address of C++ `operator[]` The libc++ locale header contained the following pattern multiple times: to get a pointer the `std::string`'s internal character array it was using the expression `&__buf[0]`. However, `std::string` has an `operator[]` that returns a `char&` and therefore should be bounded to only one byte when CheriSH is enabled. The fix for this is to use `std::string::data()`[27] instead, as this returns a pointer to the underlying storage.[28] The <filesystem> implementation also took the address of the references returned by the `std::string` `front()` and `back()` methods, expecting the resulting pointer to be bounded to the entire string. Again, this can be fixed using the `data()` accessor.

As this appears to be a reasonably common pattern, I modified the compiler to emit a warning whenever the address of an overloaded `operator[]` or a reference-returning

[26] GCC-MPX provided a `-fchkp-first-field-has-own-bounds` compiler flag to also narrow bounds for the first structure member. However, there did not appear to be an option for choosing the bounds narrowing behaviour for individual (non-array) structure members.

[27] When using C++ standards prior to C++14, there is no non-`const` `data()` accessor, so we use `str.begin().base()` to retain compatibility with code targeting older C++ standards.

[28] It might be possible to automatically translate `&str[N]` into `str.data() + N` similar to C arrays. However, this would require hard-coding types for which this transformation is safe (e.g. `std::string` and `std::vector`) and would not work for user-defined types unless we provide new annotations.

`front()`/`back()`/`at()` function is taken. Using this warning, I located the remaining cases in the <regex> header and the <filesystem> and <string> implementation source files, as well as in various libc++ test suite source files.

C++ references to the full array I also found one case in libc++ where the programmer expected the entire array bounds to be used for references. In the implementation of `std::string` a single-element reference was passed to `std::addressof()` in order to create a pointer to the full array. It is not clear why libc++ does not use the address-of operator here since this would have worked as expected with CheriSH.[29]

5.4.1.7 Other incompatibilities

While changing CheriBSD to use CheriSH I also discovered other incompatibilities and compiler bugs. However, unlike the previous issues, these do not seem to be recurring patterns.

Obtaining a one-past-the-end pointer The code in /bin/test attempts to get a one-past the end pointer using the syntax `(&array)[1]`. Without CheriSH this is equivalent to adding `sizeof(array)` to `array`, but with CheriSH this will attempt to set bounds on the resulting array decay to `sizeof(array)` and trap because it is out of bounds. I fixed this problem by adding a macro `array_one_past_end()` and use that instead of the previous strange syntax.

Multi-dimensional arrays I noticed that some code such as the JPEG benchmark in MiBench converts a multidimensional array to a plain pointer using `&array2d[0][0]`. To be consistent with `&array1d[0]` not tightening bounds in the default CheriSH mode, this should also apply for two-dimensional arrays but is currently not implemented.[30] As there have only been three cases of this pattern so far (and all in MiBench), I used `__builtin_no_change_bounds()` to work around the compiler limitation.

Taking the address of weak globals The initial implementation of CheriSH always inserted a `CSetBounds` whenever it encountered an address-of operator in C (to the size of the target type), even for global variables. This previously caused issues with undefined weak symbols since setting bounds on a `NULL` pointer will trap. I have since fixed the compiler to narrow the bounds only for defined weak symbols. These bounds on globals are not strictly necessary and do not provide any security benefit since RTLD ensures that these pointers are sensibly bounded (see Section 4.4). However, they can

[29] This issue highlighted the problem that we cannot set bounds when converting C++ references into pointers in `std::addressof()` since the dynamic type of the reference is not known (see Section 5.3.3).

[30] Changing the behaviour is possible, yet non-trivial to add to the current Clang implementation. We plan to fix this problem in future versions of CheriSH.

	Pass	Fail	Skip	Total
FreeBSD CheriABI	5373	276	577	6226
FreeBSD CheriSH	5371	278	577	6226
libc++ CheriABI	5623	13	594	6230
libc++ CheriSH	5618	18	594	6230

Table 5.2: Test suite results with and without CheriSH.

be useful to catch a mismatch between the C declaration and the actual symbol type. Additionally, other linkage models might decide not to bound DSO-local globals. As CheriSH is target-independent and implemented in the Clang frontend, we currently bound all globals. If bounding on each access turns out to have adverse performance effects, we may revisit this decision, but so far it does not appear to make any measurable difference (see Section 5.4.3).

5.4.2 Test suites

The FreeBSD test suite contains over 3500 programs, is part of the FreeBSD base system and provides tests for many programs and libraries. This test suite was useful in finding compiler bugs and incompatibilities, but even in the initial run of the test suite we encountered very few test failures compared to the CheriABI baseline.[31] The results for this test suite for both CheriABI and CheriSH can be seen in Table 5.2. Currently, there are only two undiagnosed failures for CheriSH compared to the CheriABI baseline: one related to signal handling and one `setjmp()` test.

To evaluate C++ compatibility, I ran the libc++ test suite for CheriABI and CheriSH. After fixing sub-object-related compiler warnings (see Section 5.4.1) in libc++, we encountered 59 additional test failures compared to the CheriABI baseline. However, it turned out that all these failures were due to a mismatched structure declaration in the threading library libthr.so (see Section 5.4.4). The only additional failures for CheriSH compared to CheriABI are caused by a compiler bug that is breaking some UTF-8 tests.

In addition to these tests, I use a version of CheriBSD compiled with CheriSH enabled for daily development, which has been extremely useful in finding problems with CheriSH.

5.4.3 Performance

In this section we compare CheriSH to a pure-capability baseline to see the cost of complete spatial safety compared to object-granularity protection. A performance comparison between pure-capability code and the insecure MIPS baseline can be seen in Section 6.2. To compare to the pure-capability baseline, we must consider where CheriSH adds new overheads. Since all capability-relative pointer accesses are already bounds checked (and

[31] Even though many tests failed in the initial run, almost all of these were caused by the same issue: `fts()` would crash (see Section 5.4.1.5).

sub-object accesses already use capabilities, albeit with larger bounds) we do not add any dynamic overhead to memory accesses. Nevertheless, the tighter run-time bounds do come at a run-time cost: whenever we set bounds on a sub-object, we must insert a `CSetBounds` instruction which returns a new, suitably bounded capability. This results in three factors that cause minor, yet measurable, performance overheads.

First, we must consider the cost of the `CSetBounds` instruction. While `CSetBounds` is one of the most complicated instructions in the CHERI ISA, it is still a single-cycle instruction and has a result that can be used immediately without requiring pipeline stalls. We can usually emit `CSetBounds` with an immediate operand (allowing for bounds up to 2047 bytes) as most sub-object pointers refer to small arrays, integers, pointers or small structures. If we cannot use the immediate operand, we must generate an integer value in a register and use that as an argument to a version of `CSetBounds` that takes a register operand. In this worst case, it results in a three-instruction sequence for bounds up to 4GiB. Therefore, this overhead is minimal unless it is used in a very tight loop.[32]

Secondly, the tightly bounded sub-object capabilities could result in increased register pressure since we can no longer reuse the same capability to refer to different members of the same structure. However, this case only happens when taking the address of a structure member or passing a pointer to an embedded array to another function. For example, sub-object accesses such as `x = foo->a + foo->b` still use the same register for the `foo` pointer when loading the value. Furthermore, in almost all cases where we pass a tightly bounded capability, code without CheriSH would also need to use separate registers since the pointer value needs to be adjusted. It only makes a difference if the compiler can remove the original pointer increment and use a load with an immediate offset. However, in this case we should also be able to prove that the access is in bounds and therefore remove the `CSetBounds` instruction for CheriSH.

Finally, the added `CSetBounds` intrinsics can limit the optimizations that LLVM performs. We have not yet updated all LLVM passes to handle the intrinsic so there could be some cases where it is treated as an opaque optimization barrier.

Optimizations To reduce the (already low) overhead of CheriSH, it would be possible to enable sub-object bounds only on structures that we consider being of interest to an attacker. Gil et al. use a LLVM pass to find structures containing both function pointers and data arrays [82]. Similarly, `-fstack-protector-strong` [43] limits instrumentation to functions with interesting stack layouts.[33] We could use similar techniques to omit sub-object bounds for certain structures. However, unlike conventional architectures where function pointers are of high interest, CHERI already protects function pointers from being overwritten with arbitrary data. As the most likely attack on a CHERI pure-capability

[32]If the bounds are being narrowed on the same sub-object in each iteration, the `CSetBounds` can be hoisted out of the loop, thus removing the overhead.

[33]Such optimizations can be risky: LLVM 9.0 was released with a broken `-fstack-protector-strong` after changing the analysis for when to omit stack canaries [143].

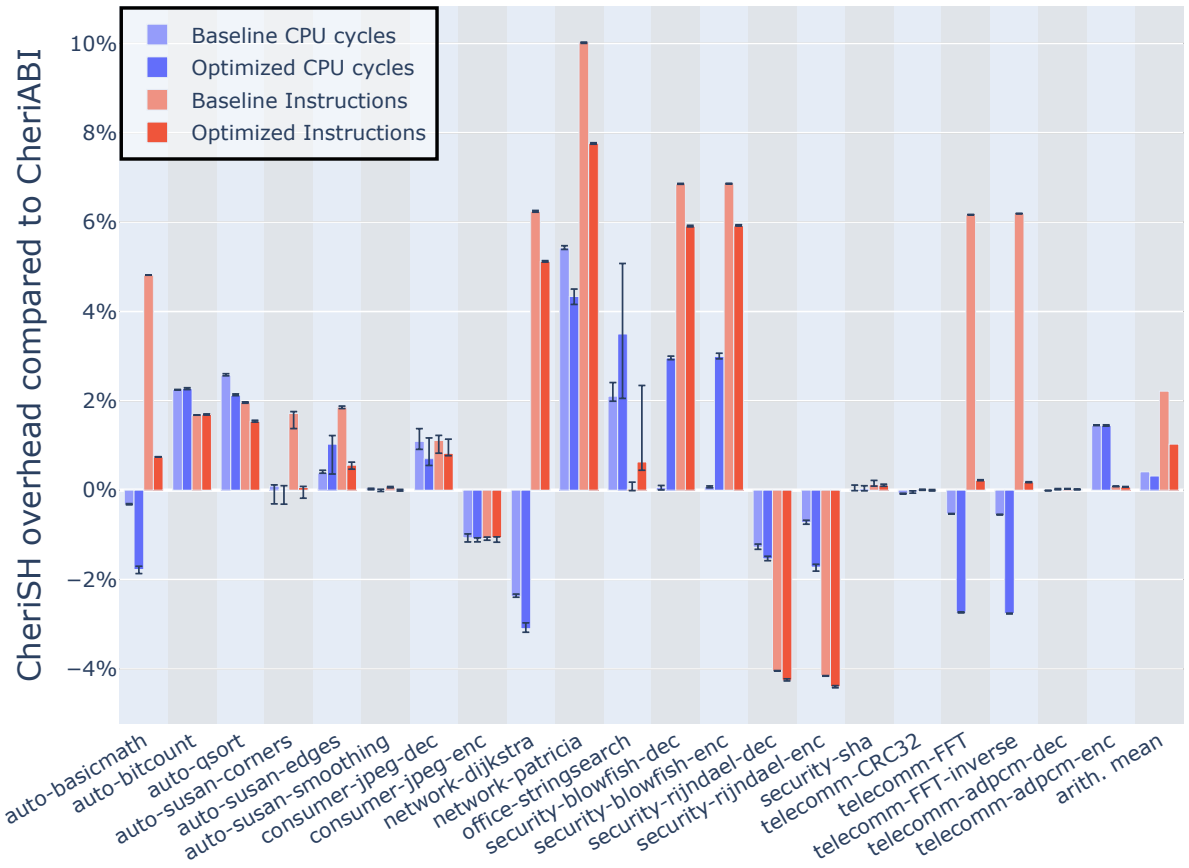


Figure 5.2: MiBench benchmark results comparing CheriSH (before and after adding optimizations) to CheriABI performance.

system is a data-only attack that corrupts adjacent data members, it seems that there is no obvious choice for structure members that do not need protecting. Nevertheless, in one case we can omit the `CSetBounds` instruction: we do not need it if the compiler knows that the access will be in bounds (e.g. due to the accesses using constant offsets only) and the bounds of the input argument are known to be larger than the new size. This optimization reuses the logic used by the stack bounds optimization (see Section 3.8.2), and therefore we can be certain that we do not impact protection.

The other optimization that we added was removing `CSetBounds` instructions with the same size. This can happen for array subscript expressions, since the initial array decay will narrow bounds and the array subscript will set the same bounds again. As `CSetBounds` is idempotent, we taught the compiler to remove multiple `CSetBounds` instructions that result in the same pointer.[34] Applying these two optimizations removed 4544 `CSetBounds` instructions from `libc.so` (30.0% of the total `CSetBounds` instructions added by CheriSH).

[34]It is also possible to remove `CSetBounds` with a larger size if the only use of that instruction is another `CSetBounds` with smaller bounds. However, this could result in differing run-time behaviour: if we remove the larger `CSetBounds`, we might optimize away a trap that would occur if the current bounds are less than the larger value.

Benchmark analysis As can be seen in Figure 5.2, the worst-case overhead for CheriSH before applying optimizations was 10.0% more instructions and 5.4% more cycles compared to the CheriABI baseline. After adding the optimizations, this overhead drops to 7.8% instruction and 4.3% cycle overhead in the worst case (network-patricia). On average (geometric mean), the benchmarks ran 0.3% more slowly and executed 1.0% more instructions. Interestingly, some benchmarks were faster with CheriSH and some even executed fewer instructions. Considering that CheriSH is an instrumentation that adds additional instructions, I assumed this was an error in the benchmark setup. However, after looking at the generated code for the benchmark where this is most visible, security-rijndael, it turns out that this is the result of CheriSH forcing the creation of bounded sub-object pointers with the correct offset. When compiling without CheriSH, these offsets are resynthesized for every array access, which adds multiple instructions.[35] As a result of this the `encrypt()` and `decrypt()` functions are 119 instructions shorter (7%) when compiled with CheriSH. Another interesting benchmark is telecomm-FFT, which executes 0.2% more instructions but runs 2.7% faster. This can be explained by different code layout and stack-frame sizes resulting in different cache utilization. CheriSH debug features (see Section 5.3.5) add up to 18% cycle and 28% instruction worst-case overhead and therefore we do not enable them by default.

5.4.4 Real issues found in CheriBSD

Existing tools such as ASan do not find sub-object underflows/overflows, so unlike per-allocation bounds checking — where ASan test coverage (and fuzzing) has already found most issues — CheriSH finds errors even in common code paths. In the process of refining the CheriSH implementation until it could compile a CheriBSD image that boots and runs tests, I discovered some real sub-object overflows in addition to finding C-programming patterns that require CheriSH opt-outs.

Buffer overreads in jemalloc and libarchive The default FreeBSD memory allocator, jemalloc [72], contained a sub-object out-of-bounds load. A loop in the bitmap code was loading the next element before checking if the current index was greater than the maximum. This is always safe in the case where I found it because the bitmap was always embedded in a larger allocated structure. However, it is loading the value one past the end, so it could crash in cases where the `bitmap_t` is not inside a containing structure. I found the same pattern in libarchive's `tar_atol_base_n()`, which loaded the next character at the end of a loop before checking the remaining count. In this case the next character is always part of the next structure field so this bug is not exploitable.

[35] This missed optimization in the LLVM MIPS backend is particularly noticeable since the current CHERI-MIPS FPGA pipeline is very sensitive to instruction bloat. The effect may be less visible on other implementations.

Out-of-bounds 2D array write in `awk` While running the FreeBSD test suite with CheriSH, we discovered that `awk` was crashing. It turns out this was caused by writing two elements past the end of a two-dimensional array. As two-dimensional arrays are contiguous, this write would almost always go unnoticed.[36] However, CheriSH sets tight bounds for each array dimension, so we caught this error. I have submitted a fix to the upstream maintainers which has been accepted [185]. This is the only issue we found with CheriSH that could also be detected by ASan plus UBSan.

Buffer overflow in `cheritest` Ironically, the testing framework used for testing of basic CheriBSD functionality (`cheritest`) contained a buffer overflow that was detected by CheriSH but not detected by CheriABI. Here the wrong field was used inside a `sizeof` expression passed to `memcpy()`. Even though the copied object is nominally fixed-size, it happened to be allocated using a call to `mmap()`. As this allocates an entire page (and therefore the resulting capability is bounded to 4096 bytes), the overflow cannot be detected by CheriABI.

Layout incompatibility of `_Unwind_Exception` Using CheriSH, we were also able to find a layout incompatibility between two declarations of `struct _Unwind_Exception`, used for thread unwinding and C++ exceptions. FreeBSD provides a header `unwind.h` that declares the basic `libunwind` functions and data types. This header uses `uint64_t` in `struct _Unwind_Exception` as specified by the ABI document [29]. However, the actual implementation is provided by LLVM `libunwind` which instead uses `uintptr_t` and explicit padding for 32-bit architectures to achieve a compatible layout. In CHERI pure-capability mode, using `uintptr_t` causes the LLVM structure to be larger than the FreeBSD one, and we were writing past the end of the allocated structure when writing to the last member. We did not discover this issue earlier because the `struct _Unwind_Exception` is embedded inside another object. Without CheriSH this means the following member is overwritten, but with CheriSH we get a bounds violation instead. Another reason that we did not discover this issue before is that changing the following member does not result in a crash and only causes incorrect stack unwinding in some rare cases. We believe that this problem would have remained undiagnosed for years if I had not created CheriSH.

5.4.5 Memory protection benefit

To evaluate the memory protection benefits of CheriSH compared to other techniques we use the BOdiagsuite suite of 291 programs by Kratkiewicz [125]. This test corpus was also used by Hardbound [58] and CheriABI [54]. Initially, I planned to evaluate CheriSH using the much larger Juliet CWE test suite [24]. However, it turns out that CheriABI catches

[36]The following structure member would only be overwritten in the unlikely case that the overflow happens when also indexing the last element in the first dimension.

all buffer overflows in this test suite, which means that it does not evaluate sub-object protection. BOdiagsuite is also not ideally suited for this evaluation since it only tests C array overflows. However, it also includes five tests that overflow into the next structure field and seven that overflow into padding. As the existing test suites barely exercise sub-object buffer overflows, I wrote many Clang tests to verify that the bounds inserted by CheriSH will prevent run-time sub-object overflows. However, these tests cannot be used by any other tool, so we rely on BOdiagsuite for this comparative evaluation.

Each BOdiagsuite test case contains one variant without a off-by-one error (*min*), a medium-sized 8-byte error (*med*) and a 4096-byte out-of-bounds access (*large*). Additionally, there is also one version of each test that touches the last accessible byte to detect false positives. These tests were designed for static analysis tools [124], so sometimes the violation was in code that the optimizer would remove (e.g. dead stores to variables not marked `volatile`). Therefore, we had to compile the tests with optimization disabled.

The original version of BOdiagsuite allocates almost all buffers on the stack and includes very few test cases for heap allocations created by `malloc()`. Most of these stack buffer overflows can be detected using stack canaries [46] and do not require more sophisticated defences. Therefore, I added versions of the original 291 test cases with all stack buffers replaced by heap allocations [47].

The memory-safety tools we compare against are ASan [201], stack canaries [46], `_FORTIFY_SOURCE` [27, 115], Valgrind [164, 165, 166], EffectiveSan [63], SoftBound-CETS [156, 160, 161] and finally CheriABI [54] as a baseline for CheriSH. Besides checking whether the buffer overflows are detected, we also verify that the variants without memory-safety errors run correctly.

Compiling and running the test programs as CheriABI catches almost all errors. However, 12 test cases include intra-object overflows which are found only when CheriSH is enabled (both in the safe and aggressive mode). None of the other tools can detect all overflows (see Table 5.3).

Stack-protector works well at detecting the *min* and *med* stack overflows since the buffers are located near the stack canary. However, for the large overflow many of the writes skip over the canary (and the return address) and only overwrite unused stack memory, thus not resulting in a crash and not being detected.

ASan and Valgrind use per-object metadata to enforce spatial and temporal safety. This design choice prevents them from enforcing complete spatial safety, which would require per-pointer metadata [209]. There are at least three cases of bounds violations that these tools cannot detect (two of which are tested by BOdiagsuite). Firstly, a memory access that is sufficiently out-of-bounds can skip the red-zones between objects and reach another valid object and is therefore not detected. This problem can be observed in the large stack overflow case for some BOdiagsuite tests with GCC ASan and Valgrind. Secondly, overflows from one global object to another are not detected. CheriABI and CheriSH detect this case since RTLD sets appropriate bounds for every global (see Section 4.4). This case

	stack			heap		
	detected	crashes	good	detected	crashes	good
Linux (insecure)	0/0/0	3/3/52	**291**	0/0/0	2/2/5	**291**
Linux (sp)	246/254/0	2/2/146	**291**	0/0/0	2/2/5	**291**
macOS (insecure)	0/0/0	0/1/274	**291**	0/0/0	0/0/2	**291**
macOS (fortify)	12/12/11	0/1/274	**291**	0/0/0	0/0/2	**291**
macOS (sp)	247/255/0	0/0/274	**291**	0/0/0	0/0/1	**291**
macOS (sp+fortify)	259/266/11	0/0/274	**291**	0/0/0	0/0/2	**291**
Valgrind 3.15	34/34/283	**0**	**291**	278/288/289	**0**	**291**
Valgrind 3.13 (sp)	259/263/125	**0**	**291**	278/288/289	**0**	**291**
Linux GCC ASAN	276/285/124	**0**	**291**	276/286/288	**0**	**291**
macOS ASAN	277/286/286	**0**	**291**	277/287/**290***	**0**	**291**
EffectiveSan 0.1.1	267/277/270	**0**	**291**	267/277/279	**0**	**291**
SoftBoundCETS 3.9	272/282/283	**0**	**290**	272/282/283	**0**	**290**
CheriABI	279/289/**290***	**0**	**291**	279/289/**290***	**0**	**291**
CheriSH (safe)	**291**/**291**/**290***	**0**	**291**	**291**/**291**/**290***	**0**	**291**
CheriSH (aggressive)	**291**/**291**/**290***	**0**	**291**	**291**/**291**/**290***	**0**	**291**

* The maximum passes in the *large* overflow case is 290 since one test cannot be run.
(sp) Tests were compiled with `-fstack-protector-all`.
(fortify) Tests were compiled with `-D_FORTIFY_SOURCE=2` (we were not able to test the GLibc/GCC `_FORTIFY_SOURCE` implementation as it requires compiling with optimizations enabled).

Table 5.3: BOdiagsuite results showing the detected errors for the *min/med/large* cases out of 291 total tests. The *detected* column lists failed runs detected by the tool and the *crashes* column contains the number of unrelated failures (such as segmentation faults). Linux tests used GCC 9.3 and macOS used Apple LLVM 10.0.1.

is similar to the first one but is not tested by BOdiagsuite. However, in Section 6.1.4 I present an example of such an overflow in libFuzzer that has not been detected previously. Finally, overflows within the same allocation (e.g. adjacent buffers within a C struct) cannot be detected as these are all part of a valid allocation.

EffectiveSan should be able to detect sub-object overflows, however, for these tests it seems that limitations of the underlying Low-Fat pointers scheme [64, 65] result in some of the small overflows not being detected. Moreover, EffectiveSan assumes that files are compiled with optimizations, so compiling without may cause some analyses to fail.

SoftBoundCETS was the only tool that included false positives: one test (basic-00013-ok) reported a memory-safety violation due to `shmget()` not being handled correctly by the SoftBoundCETS runtime. Another test (basic-00182) resulted in a segmentation fault (or sometimes an infinite loop) for both the known good test case and all overflow variations because the interceptor for `fgets()` was broken. This clearly shows that the libc function-interception strategy does not scale (see Section 5.4.7). While the SoftBoundCETS paper

suggest that sub-object protection is supported [160], I could not find a way of enabling it,[37] and therefore SoftBoundCETS can only find 272 of the 291 memory errors.

The BOdiagsuite results match the findings of Oleksenko et al. who evaluated memory protection using the RIPE buffer-overflow test suite [252] and showed that neither ASan nor SoftBoundCETS could protect against sub-object overflows [174]. I could not use this test suite for CheriSH since it is x86-only and none of the exploits would work for CHERI. However, based on the MPX results [174], we can assume that CheriSH would have prevented all exploits.

5.4.6 Implementation complexity and maintainability

By building CheriSH on top of CHERI pure-capability code generation, I was able to implement CheriSH entirely in the Clang compiler frontend without requiring any changes to the LLVM backend. The code to determine whether bounds should be tightened or not is self-contained and is currently 897 lines of code in `CGExpr.cpp`. For each of the four expressions identified in Section 5.3, I modified at most two functions to call this new bounds-tightening code. Additionally, I also introduced two new attributes which adds 48 lines across three files. The change that modified the most files was the introduction of `__builtin_no_change_bounds()` since it required adding a new type of expression to the Clang abstract syntax tree (AST). Therefore, this change was one of the largest, touching 23 files with 336 insertions and 51 deletions. However, it should not introduce any maintenance burden as the new expression node (the class `NoChangeBoundsExpr`) inherits from the `ParenExpr` class, which is an AST node representing a parenthesized expression. In terms of C/C++ semantics, `__builtin_no_change_bounds()` has the same effect as surrounding an expression with parentheses (except in cases where CheriSH would add sub-object bounds). Therefore, the only place in the Clang source code that needs to explicitly handle this new AST node is the CheriSH code in `CGExpr.cpp` and because of inheritance almost all code in Clang automatically treats the new node in the same way as parentheses.[38] Finally, I also added around 3500 lines of tests. As these are new files, they do not cause any maintainability issues. We have been maintaining this code for over one year and have not yet seen any merge conflicts or compilation failures in the `git` merges performed since.[39]

5.4.7 Protection without library interception

Some memory-safety tools rely on interceptors/wrappers for common functions (e.g. `memcpy()` in libc) to model the memory accesses of these functions and update metadata.

[37] Looking at the released source code [155], sub-object protection does not appear to be included.

[38] Due to the way that LLVM implements dynamic casts, I had to update a few switch statements and AST visitor callbacks to handle the new expression node.

[39] The last merge from upstream LLVM was performed on 13th September 2019.

Implementing these wrappers can be error-prone and result in false-positives or negatives [174, 216, 264]. Zhivich and Leek noted that the wrappers for `sscanf()` and `fscanf()` were incorrect when they evaluated CCured [264]. Moreover, I discovered that the SoftBoundCETS wrapper for `fgets()` was crashing (see Section 5.4.5). I also noticed that the SoftBoundCETS runtime contains a wrapper for `qsort()` to retain bounds metadata when moving elements. However, there is no wrapper for `qsort_r()`, so any call to that function will corrupt bounds metadata and result in incorrect bounds information being used for subsequent calls. As SoftBoundCETS is a research prototype and not a production system, such missing wrappers are not unexpected. Nevertheless, even ASan, a very widely used tool, suffers from problems with interceptors. While comparing the memory protection benefits of CheriSH with ASan, I noticed that LLVM 8.0 (the current version when this dissertation was written) shipped with a completely broken ASan on FreeBSD. The error was caused by interceptor functions used in early start-up delegating back to the intercepted function, thus resulting in infinite recursion. I have since committed a fix to LLVM and version 9.0 will again have a working ASan on FreeBSD.

These problems are not an exception: MPX (which could be considered a production system) also suffered from issues with interceptors. Oleksenko et al. report that the MPX interception runtime had bugs in `memcpy()` and did not catch a stack overflow in NGINX due to a missing wrapper for `recv()` [174].

All these issues illustrate that interception is fragile and is prone to break especially for less widely used libraries or operating systems.[40] We therefore believe that systems that are not just used for debugging must work without interception. Unlike most other techniques, CheriSH enforces sub-object bounds — even in code that has been compiled without CheriSH — and most importantly does so without the need for wrapper functions.

5.5 Future work

While we believe that CheriSH is a mature system that could be deployed with only minimal changes, there are some additional features that might be useful to add.

Sub-object protection by default CheriSH is currently not enabled by default, but we would like to change this in the future. However, we do acknowledge that there is a porting effort — albeit a very small one — for some projects to be compatible with sub-object provenance. Therefore, this decision will have to be evaluated carefully.

Targeted fuzz-testing with CheriSH We have started doing some fuzz-testing with CheriSH, but so far we have only found inter-object overflows and no sub-object ones.

[40] Yet even for commonly used operating systems, it is difficult to accurately model memory effects of rarely used, multiplexing system calls such as `ioctl()`. In contrast, for CheriSH the CheriABI system-call layer ensures the kernel adheres to userspace capability permissions [54].

We believe that we can find many issues by specifically fuzzing libraries that use buffers embedded in structures. These could easily be identified with small compiler modifications similar to the one used by Gil et al. to find vulnerable data structures [81, 82].

Strict bounds for C++ class hierarchies and `container_of` In the current design, we do not tighten bounds when upcasting since base classes might call virtual methods that expects access to the subclass fields. In the future, we would like to support this narrowing of bounds. As the monotonicity imposed by CHERI prevents widening of bounds when downcasting, we could store a pointer to the entire object next to the *vtable* and use this pointer to rederive an appropriately bounded capability. This would not protect from a sufficiently capable malicious attacker, but it would prevent accesses to memory that is not part of the current static type via the `this` pointer, which would allow detecting some cases of type confusion. A similar approach could be used for `container_of` but would require source-level changes to load the larger capability.

The capability spanning the entire object could be sealed and only be usable by a more trusted 'downcasting/`container_of`' component. This should provide security at the cost of a privileged function call.

Improved opt-out annotations While we already provide quite fine-grained annotations for opting out of sub-object bounds (see Section 5.3.2), we could still reduce the amount of unnecessary privilege by adding even finer-grained annotations. For the C++ inheritance in C pattern, we could add a new attribute to annotate the 'base class' type to avoid setting bounds when this type is used as the first member. This requires fewer changes than annotating the field in each subclass and more importantly still allows setting bounds for `foo->first_member->buffer`.[41] We could also provide finer-grained control over when the `cheri_no_subobject_bounds` attribute should prevent bounds narrowing (e.g. only for the address-of operator or only for C++ references). Finally, for the `container_of` opt-out, we could add an annotation to use the size of the containing structure instead of the remaining allocation size.

5.6 Related work

Systems that enforce spatial safety at a sub-object level have been developed before. However, these approaches often incur high performance overhead or have poor compatibility with existing code. In this section I present some techniques that explicitly consider sub-objects. This is by no means an exhaustive list as many memory safety techniques that *could* protect sub-objects have been proposed in the past — for example, most techniques that use per-pointer bounds metadata.

[41] This is not currently the case since the `cheri_no_subobject_bounds` annotation also applies to all fields of the annotated member.

AddressSanitizer Some (albeit very limited) support for bounds checking below allocation granularity exists in ASan. Firstly, ASan can be paired with the UBSan option `-fsanitize=array-bounds`, which allows it to detect sub-object overflows for array subscripts on fixed-size arrays (see Section 5.3). Secondly, for code such as `std::vector`, ASan users can call `__sanitizer_annotate_contiguous_container()` to highlight which parts of the allocation are currently accessible. As the whole `std::vector` region is allocated by `malloc()`, the entire region is initially marked as valid for accesses. Using this hook, subsets can be flagged as invalid and will trigger faults when accessed.

It may be possible to use this technique to achieve some degree of sub-object safety by marking everything else in the object as inaccessible. Therefore, if multiple sub-objects are in use at the same time, this would need to call this hook before every sub-object access. Furthermore, this approach breaks when passing two sub-objects to an external function as they would both have to be marked as accessible for the entire duration of the call.

It seems unlikely that ASan could be modified to automatically add sub-object bounds. The ASan metadata only marks *memory regions* as being accessible and does not associate access permissions with *pointers*. Therefore, it cannot differentiate a sub-object overflow from a valid access using a pointer to another structure member. Moreover, ASan incurs high run-time overheads, requires *all* code to be instrumented and does not support static linking due to libc interception. In contrast, the sub-object bounds set by CheriSH are architecturally enforced even for code compiled without CheriSH.

MemSafe This spatial and temporal memory-safety tool by Simpson and Barua [209, 210] relies on compiler-inserted software checks before memory accesses to detect spatial and temporal errors. MemSafe uses a hybrid metadata approach with metadata for every pointer as well as for every object. Due to this approach it is possible to provide both temporal safety (using object metadata) and sub-object spatial safety (using pointer metadata). While MemSafe has support for bounding sub-objects, there does not appear to be any support for customizing this behaviour and when taking the address of an array element it will always use the full array bounds. Moreover, the authors were only able to compile a subset of the SPEC benchmarks with their instrumentation whereas CheriSH can boot a full operating system and run complex applications.

EffectiveSan '*EffectiveSan*' by Duck and Yap [63] also protects sub-objects (using type metadata) in addition to protecting object bounds (using Low-Fat pointers [64, 65]). The analyses and checks performed by EffectiveSan are more detailed than those provided by CheriSH since they take the run-time type into account, but therefore also come at a very high performance overhead (e.g. 1200% for some Firefox benchmarks). Besides the performance overhead, another downside compared to CheriSH is that EffectiveSan is a compiler instrumentation that relies on all memory accesses being intercepted, so any

uninstrumented code breaks the provided guarantees. Unlike CheriSH, EffectiveSan is a debugging tool and not a security feature.

HardBound, SoftBound and Watchdog '*Hardbound*' provides hardware support for bounded pointers. The paper claims to set bounds for sub-objects [58, §3.2], and mentions that their compiler does not attempt to narrow bounds for arrays due to single-element/whole-array ambiguity (see Section 5.3.3). Additionally, HardBound pointers do not provide monotonicity, which could allow malicious code to widen bounds.

A later software-based technique, '*SoftBound*' [160] (with an overlapping set of authors), mentions further caveats related to sub-objects: the paper notes that `container_of()` (see Section 5.4.1.1) and taking the address of an array element may cause compatibility problems. The same caveats are also mentioned in a 2015 paper describing SoftBound-CETS [156] (a combination of SoftBound and CETS [159]). They state that they did not encounter any sub-object incompatible patterns in the '23 benchmarks (approximately 272K lines of code)' that were evaluated. However, they conclude that sub-object bounding should be an opt-in feature as it could break existing code. In the PhD dissertation describing SoftBoundCETS [161], we find the same discussion of sub-object bounds, yet looking at the source code of the instrumentation [155], there does not appear to be any support for sub-object protection. Additionally, SoftBoundCETS is implemented as an LLVM pass, rather than being integrated with Clang, so it lacks C-language type information to perform accurate sub-object bounds checks.[42] SoftBoundCETS also does not work with multi-threaded code [176], has false positives [215] (see Section 5.4.5) and does not support integer-to-pointer casts [215], which are common in many C and C++ programs. Moreover, it was only tested on a small set of benchmarks instead of a full operating-system userspace.

The later '*Watchdog*' [157] and '*WatchdogLite*' [158] systems are similar proposals (in hardware) by the SoftBoundCETS authors that could also theoretically protect sub-objects. Based on the SoftBoundCETS results, the results in Section 5.4.5 and those reported by Oleksenko et al. [174], it appears that while these systems could be modified to enforce sub-object protection, they do not actually implement it.

__FORTIFY_SOURCE This technique can detect some sub-object overflows when compiling in the strictest mode, in which instrumented functions (e.g. `memcpy()`) check whether the passed size exceeds the sub-object size. The `_FORTIFY_SOURCE` mechanism relies on the `__builtin_object_size()` compiler built-in [80]. Support for sub-object protection has been present since the original patch that added `_FORTIFY_SOURCE` to GCC: 'The diff[er]ence between `-D_FORTIFY_SOURCE=1` and `-D_FORTIFY_SOURCE=2` is e.g. for `struct S { struct T { char buf[5]; int x; } t; char buf[20]; } var;` With `-D_FORTIFY_SOURCE=1`, `strcpy(&var.t.buf[1], "abcdefg");` is not con-

[42] Some LLVM passes remove type information or replace structures with byte arrays of the same size.

sidered an overflow (object is whole VAR), while with `-D_FORTIFY_SOURCE=2` `strcpy(&var.t.buf[1], "abcdefg");` will be considered a buffer overflow' [115].

If a fortified function reads or writes past the end of the object, an error is raised and the program terminates. However, this approach requires wrappers for each library function that should detect these buffer overflows. Additionally, it can only compute object sizes for pointers where the compiler sees the creation (e.g. when using the address-of operator), so wrappers that forward pointers to a fortified function prevent overflow detection.[43] Finally, the `__builtin_object_size()` built-in only works reliably at higher optimization levels, so this instrumentation is not effective in debug builds.

One advantage of the `_FORTIFY_SOURCE` approach is that some errors can be detected at compile time, whereas CheriSH and ASan will only catch the problem when it occurs at run time [27]. However, these compile-time bugs can also be found using a static analyser, which would not require adding wrappers for all library functions.

Intel MPX Intel Memory Protection Extensions (MPX) are an ISA extension for x86 that adds bounds-checking instructions. Oleksenko et al. provide a detailed analysis of Intel MPX including performance measurements and compatibility concerns [174, 175]. Similar to ASan and SoftBoundCETS, MPX also relies on intercepting system library functions to model the updates to bounds metadata (using the runtime library libmpxwrappers). Moreover, bounds checks must be inserted explicitly, so any access without a prior bounds-checking instruction will succeed. Furthermore, the requirement for correctly inserted bounds checks means that compiler becomes part of the TCB (in addition to the MPX runtime and the complex Intel x86 hardware) whereas for CheriSH we only need to trust the much simpler CHERI hardware (and not necessarily the compiler) since we can rely on monotonicity and pointer integrity. This means that even though this is a hardware-enforced mechanism, it is optional and (unlike CheriSH) cannot enforce protection for uninstrumented code. Any pointer modified or returned by uninstrumented libraries will have the bounds nullified (or set incorrectly), thus bypassing all MPX checks and effectively preventing interoperability with other libraries [174].

Another large downside of MPX is high performance overheads. Even though it is a hardware mechanism, it incurs a mean overhead of over 50% for SPEC2006 (similarly to the software-only ASan) [174]. Moreover, bounds metadata is created on-demand by handling traps and therefore must be managed by the kernel. In certain micro-benchmarks this can cause 130% overhead [174].

Despite these downsides, MPX can be used to enforce sub-object protection. Since the bounds are associated with each pointer, narrowing bounds for sub-objects is possible and was done by default in GCC. However, MPX fails to compile and run many projects due to compiler bugs and limitations (e.g. no support for C99 variable-length arrays) or produces

[43] Recent versions of clang support a `pass_object_size` attribute that can forward the object size though multiple levels of wrapper functions, but each of these functions needs to be modified manually.

incorrect bounds at run time [174], resulting in poor compatibility and low adoption rates for existing code.

Finally, support for MPX was included by default in GCC but has been removed in version 9 [76]. Linux kernel support was added in version 3.19, but removal of this code started with version 5.4 [154]. This indicates that Intel might remove MPX from future Intel CPUs since it has not gained much adoption.

Memory Safe C dialects Some memory safe C dialects provide the ability to protect sub-objects. For example, '*Checked C*' [67] extends the C type system with new pointer-like types that have bounds. The paper states that sub-object protection is enabled: 'When taking the address of a struct's member `(&p->f)`, the bounds are those of the particular field. On the other hand, the address of an array element retains the bounds of the whole array.' [67] However, they also state that fully porting code to use the new safe constructs would require changing between 9.8 and 35.3% (geometric mean 17.5%) of all lines of code for benchmarks that they ported. It also appears that bounds checked C implementations such as Safe C [16], Patil and Fischer [178] and Fail-Safe ANSI C [172, 173] could be used to enforce sub-object bounds but it is unclear if and to what extent they do. '*CCured*' can also support certain cases of sub-object checks by using checked casts with run-time type information (RTTI) and fat pointers [162].

Comparison to CheriSH Approaches that detect spatial violations using bounds metadata associated with objects are not able to detect sub-object overflows by design (as the object bounds span *all* sub-objects). In contrast, CheriSH includes all required metadata in the pointer and can therefore detect sub-object overflows. Additionally, most memory-safety tools rely on compiler-inserted checks to ensure spatial safety. If an attacker were to be able to inject arbitrary code, this would not include these software checks and therefore the safety constraints could be violated. It also means that passing a bounded value to uninstrumented code[44] will allow this code to bypass all spatial and temporal safety measures. This problem could be solved using a hardware-based approach, but HardBound appears to allow non-monotonic pointer manipulation and MPX has a fail-open policy, requires a large TCB and is not well-supported. While CheriSH does rely on compiler instrumentation to insert the `CSetBounds` instruction (which reduces the available bounds), we do not need to insert software checks for *any* load or store instructions. Additionally, ASan, CCured, MPX and SoftBoundCETS (and others) require wrappers for `libc` library functions to work correctly. Relying on interception is fragile and error-prone (see Section 5.4.7). By building upon CHERI, we avoid these problems and can ensure that even untrusted code must adhere to the bounds and provenance validity restrictions imposed by our runtime.

[44]This is only possible if the instrumentation is binary compatible. Approaches that change the pointer representation or structure layout cannot interact with unmodified libraries.

5.7 Summary

In this chapter I have presented CheriSH, a mechanism to enforce sub-object bounds in large C and C++ codebases. CheriSH enforces *complete* spatial safety, and the underlying CHERI architecture guarantees pointer integrity and monotonicity, thereby preventing even uninstrumented code from violating this property. To our knowledge, this is the first time bounds and provenance have been enforced at the sub-object level for an entire operating-system userspace and kernel, providing stronger spatial safety guarantees than any other existing memory protection mechanism. Moreover, unlike most other related systems, CheriSH also considers protection for C++ references and handles opportunities for bounds-tightening other than the address-of operator. I have further shown that — contrary to initial concerns — it is possible to enforce sub-object bounds in most C and C++ programs without any changes to the source code. Additionally, I showed that the number of changes is limited to a few opt-out annotations in certain cases. The majority of these incompatibilities appear in code that was written many decades ago (before the C99 standard was well-supported in compilers) or in low-level code such as operating-system kernels. Our experience shows that the more modern the code, the more likely it is to work unmodified with CheriSH enabled (even at higher protection levels). In order to support all these cases, I have created a sub-object protection mechanism that can be applied at various levels of aggressiveness, trading compatibility for tighter run-time bounds of sub-objects. While implementing CheriSH, I also found two (benign) sub-object out-of-bounds loads in well tested software (jemalloc and libarchive) that could not have been detected by any other existing memory protection tool for C/C++. CheriSH was also able to find a (potentially exploitable) out-of-bounds write in awk and a structure layout incompatibility that we would not have been able to find otherwise. Furthermore, I have created a protection model that is resilient to untrusted or even malicious code (by virtue of being based on a strong model, pure-capability C/C++, rather than a weak model such as MPX), trusting a much smaller TCB than any other system we are aware of. Finally, I have given a classification of certain C idioms (and non-idioms) that are not compatible with sub-object provenance.

6 EVALUATION

This chapter evaluates two crucial considerations in the potential use and adoption of pure-capability CHERI: compatibility and performance. In Section 6.1, I analyse source-level changes required for CHERI pure-capability support. I find that pure-capability C/C++ is almost entirely source compatible with existing code, especially when considering the refinement of CHERI C/C++ semantics described in Chapter 3. In Section 6.2, I explore the overall performance of pure-capability code, demonstrating performance comparable to a MIPS baseline using 64-bit integer pointers when using the PC-relative CHERI linkage model (see Chapter 4). In this evaluation I focus on pure-capability C/C++ without CheriSH as the performance and compatibility impact of extending pure-capability C/C++ with CheriSH has already been measured in Section 5.4.

6.1 C/C++ source-code compatibility

One of the most important objectives for pure-capability C/C++ is compatibility with existing codebases. Evaluating the source-level compatibility improvements made through my refinements to pure-capability C/C++ is therefore a key focus of this chapter. I evaluate the current state of compatibility and the impact of these improvements based on various open-source case studies.

6.1.1 Methodology

To evaluate source-level compatibility, I analysed the changes (number and nature of lines and files modified) that we made to various projects by comparing the unmodified version to the latest CHERI-compatible `git` commit at the time. I collected this data using `cloc` [51] with the `--count-and-diff` flag and include all changes made to C, C++ and assembly files. For more realistic numbers, I exclude automatically generated source files that happen to be checked into revision control. This methodology may result in selection bias by only choosing software that currently runs on CheriBSD. To address these concerns, we ported a broad and varied corpus of software to pure-capability C/C++, including small command-line programs, relational databases, dynamic language runtimes, web browsers and entire operating-system kernels. This selection of software includes representative candidates for both legacy and modern codebases. Additionally, I chose programs that include extensive test suites so we can be certain that not only compilation

succeeds, but the programs also work as expected. Nevertheless, few test suites exercise all code paths, so we may have missed certain corner cases in this analysis.

For most projects, the porting effort was performed before I introduced refinements such as the *address* interpretation of capabilities, so many changes are in fact no longer necessary. In Table 6.1 these semantic refinements are highlighted by the check marks and quantified (using manual inspection of the changed code) in the *Notes* column. I could not perform this analysis for all projects, since manually inspecting all changed lines would have taken many months.

6.1.2 Overview

As can be seen in Table 6.1, pure-capability C/C++ is almost entirely source compatible with existing code. For most projects we had to change less than 0.1% of source lines of code (SLOC). Yet even for low-level libraries such as libFuzzer, libc or the FreeBSD kernel, the changes required to support pure-capability C/C++ are modest: at most 1.81% of SLOC. The only cases where significant effort is required are heavily optimized language runtimes such as JavaScript engines. For these runtimes, specific knowledge of pointer representations is leveraged (e.g. *NaN-boxing* [93]) to speed up execution.

As we also made changes to most projects that are not strictly required for pure-capability C/C++ (or no longer required due to the compatibility refinements made as part of this dissertation), this data represents an *upper bound* for the number of changes. For example, in the libc++ test suite the majority of changes was related to increasing timeouts for our simulation environment and for libFuzzer the majority was not required for compatibility, but was made to compile at the highest warning level and fix performance inefficiencies caused by excessive `uintptr_t` use. Except for LLVM's compiler runtime library, compiler-rt, we ported all projects listed in Table 6.1 before the introduction of the *address* interpretation of pure-capability C/C++ (see Section 3.6) and all but ICU4C (a library for Unicode handling that is required by WebKit) include capability-offset-specific changes that have become unnecessary with our refined pure-capability semantics. Similarly, many added casts can be removed as we no longer include the tag (capability validity) bit in pointer comparisons (see Section 3.2.5).

The most important observation is not visible from these numbers: most projects include multiple libraries and programs, and many of them work completely unmodified when compiled as pure-capability binaries. Out of the nearly 800 programs in the FreeBSD source tree, 765 work *without any changes*. Even for the system libraries (which usually require more changes [54]), 159 out of 212 do not include any CHERI-specific changes.

In our recent work on CheriABI [54], we showed that the FreeBSD programs and libraries can be ported to CHERI with very few changes. However, a considerable number of the changes listed in the paper are no longer necessary after switching from interpreting `uintptr_t` as an address rather than an offset from the capability base (see Section 3.6).

Project	Languages	Total counts SLOC	Files	Total CHERI changes SLOC	Files	UIntPtr	Offset	PtrCmp	CheriSH	Other	Notes
OpenSSH	C	102K	396	0 (0.00%)	0 (0.0%)						
SPECINT2006	C, C++	258K	466	0 (0.00%)	0 (0.0%)						
libxml2	C	231K	184	0 (0.00%)	0 (0.0%)						
rsync	C	41K	105	3 (0.01%)	2 (1.9%)					✓	Only required change is a bug-fix
SQLite	C	201K	300	20 (0.01%)	2 (0.7%)						
OpenSSL	C, ASM	290K	952	42 (0.01%)	3 (0.3%)		✓		✓		
PostgreSQL	C	739K	1,945	803 (0.11%)	96 (4.9%)		✓			✓	> 50% changes non-essential
NGINX	C	132K	334	145 (0.11%)	26 (7.8%)	✓	✓	✓			≈ 50% changes non-essential
FreeBSD kernel (hybrid)	C, ASM	2,457K	7,690	29,640 (1.21%)	485 (6.3%)	✓	✓	✓		✓	
FreeBSD kernel (pure)	C, ASM	2,440K	7,651	34,105 (1.40%)	590 (7.7%)	✓	✓	✓	✓	✓	
FreeBSD libc	C, ASM	211K	1,890	3,199 (1.51%)	195 (10.3%)	✓	✓	✓	✓		
ICU4C	C++, C	534K	1,553	17 (0.00%)	5 (0.3%)					✓	No CHERI-specific changes
QtBase	C++, C	1,504K	5,116	530 (0.04%)	51 (1.0%)	✓	✓	✓		✓	
QtWebkit	C++, C	1,665K	13,177	1,661 (0.10%)	106 (0.8%)	✓	✓	✓		✓	Many changes for split register file
libc++ (lib only)	C++	114K	227	133 (0.12%)	17 (7.5%)		✓		✓		> 20% changes for CheriSH
libc++ (test suite)	C++	429K	6,328	651 (0.15%)	195 (3.1%)		✓		✓	✓	> 60% changes non-essential
compiler-rt	C++, C, ASM	130K	1,066	2,359 (1.81%)	217 (20.4%)	✓		✓			

FreeBSD kernel For the CheriBSD kernel we excluded drivers to avoid under-reporting our changes as we do not use or compile all of them.

Table 6.1: Summary of changes required to run existing software in CHERI pure-capability mode. These numbers were measured with `cloc` [51] and are an upper bound. Many projects include changes that are not required for CHERI compatibility or are no longer necessary after refining our model of pure-capability C/C++. The non-essential changes are as follows: **UIntPtr**: Changes removing `uintptr_t` for improved efficiency. **Offset**: Changes caused by *offset* interpretation of capabilities. **PtrCmp**: Changes caused by including tag bit in comparisions. **CheriSH**: Changes to support sub-object bounds. **Other**: Changes such as fixes for FreeBSD compilation.

Furthermore, I have also since added many compiler diagnostics to ensure that compatibility issues are found at compile time rather than at run time. We believe that the number of compatibility issues found only at run time will drop further in the future as the compiler diagnostics become more powerful.

For many projects, the biggest challenge was not related to CHERI compatibility, but caused by build systems and test suites that are not friendly towards cross-compilation (NGINX being the worst offender here). Therefore, a significant by-product of my PhD has been the creation of `cheribuild` [186], a script that can be used to build, run, test and benchmark the projects listed above (and many more).[1] It is used by many project members and is now the recommended way to get started with CHERI software development.

6.1.3 Case studies

The following subsections explore pure-capability CHERI C/C++ compatibility based on various case studies. For most case studies, I only provide a brief overview, but I selected a few representative and interesting software packages out of the larger software corpus that I analysed. Additionally, I focus on important common themes rather than attempting to enumerate all issues. However, I analyse one particularly interesting project that I ported, LLVM libFuzzer, in greater detail in Section 6.1.4.

6.1.3.1 CheriBSD userspace

We have made changes to run all of CheriBSD (a complete adaption of FreeBSD for CHERI) userspace as pure-capability programs. In doing so, we discovered various problematic C programming patterns that made us change the original pure-capability design [41] and further refine the C/C++ semantics as has been described in Chapter 3. This porting process started before we had added compile-time compatibility checks, so compatibility issues were generally found at run time through hardware traps. The debugging process used to be very labour-intensive[2] so added compiler diagnostics significantly improved our productivity. This case study also shows the advantage of *address* interpretation (see Section 3.6) of CHERI capabilities: if we look at the 205 userspace files modified for CheriABI [54], 62 (i.e. 30%) of them include changes that are no longer necessary with the *address* interpretation.

[1] So far I have made over 2800 commits to this project, and it currently has support for building 260 targets with many configuration options.

[2] We have only had a working debugger (GDB) since 2018, when John Baldwin taught it to understand CHERI capabilities. Before then, the only debugging tools we had available were instruction-level tracing and `printf()` debugging. It cannot be overstated how much the availability of stack traces and breakpoints has improved my efficiency at debugging CHERI compatibility issues. I estimate that without GDB my port of libFuzzer (see Section 6.1.4) would have taken several weeks rather than a few days.

6.1.3.2 CheriBSD kernel

Currently, the CheriBSD kernel is compiled as a hybrid binary (i.e. all pointers are integers unless they are annotated with `__capability`). Hybrid mode generally requires more changes than compiling in pure-capability mode, as every capability use must be annotated. Therefore, we currently have changes to 6.3% of all files with 1.21% of SLOC modified (0.57% when including drivers). These changes are essential for the pure-capability run-time environment and include many features such as context-switching, swapping and the system call handling. A more detailed explanation of these changes can be found in [53].

More recently, we have also been exploring a pure-capability version of the FreeBSD kernel. Over the course of the past few years, Alfredo Mazzinghi has been working on porting the CheriBSD kernel to run in pure-capability mode. There are three interesting observations to be taken from his work. Firstly, he reports that around 10–15% of the changes he had to make to the CheriBSD kernel were related to using capability offsets as the `uintptr_t` interpretation (see Section 3.6). This is another confirmation that *address* interpretation of capabilities significantly improves compatibility with existing C code. Secondly, this shows that the CHERI linkage models (see Chapter 4) can successfully be applied to an operating-system kernel. Currently, the pure-capability kernel uses the PLT ABI (see Section 4.2.5). Finally, he was able to turn on CheriSH sub-object protection and boot the operating system with only few changes to the kernel source code (see Section 5.4.1).

6.1.3.3 NGINX

NGINX is a widely used web server written in C. The initial port of NGINX only required changes to a few files since NGINX already uses `uintptr_t` extensively.[3] In order to use NGINX as an HTTP server, we had to modify less than 100 SLOC.[4] I later fixed many more issues flagged by the new compiler warnings (especially bitwise operations on `uintptr_t`). While doing so, I also discovered an extremely odd re-implementation of `memcpy()`. In `ngx_http_log_module.c` NGINX implements `memcpy()` by bitwise-`OR`-ing and shifting individual bytes into an `uintptr_t` field and extracts them again using bitwise-`AND`s later — instead of using `memcpy()` for both cases. This optimization may have made sense many years ago, when compilers were not able to inline `memcpy()` for small sizes, but for CHERI this is less efficient since we need to constantly move between capability and integer registers to perform these operations.[5] As the bitwise variant does not preserve tags, I replaced the custom logic with `memcpy()`.

[3] In fact too extensively: the types `ngx_int_t`, `ngx_uint_t` and `ngx_flag_t` are defined as `uintptr_t` or `intptr_t`. While this is not a problem for CHERI, it is less efficient and increases memory usage by using 16 bytes for integers instead of 8.

[4] The bigger challenge was debugging unpredictable run-time crashes caused by bugs in the old `ld.bfd` linker that we were using at the time.

[5] By teaching more LLVM passes about the semantics of CHERI capability intrinsics, it might be possible to determine that this pattern is a `memcpy()`, but as of now the compiler cannot infer this.

In total, we changed 145 SLOC in NGINX. However, we ported NGINX before the *address* interpretation of capabilities. Looking at the `git` difference, it appears that at least half of the changes are no longer required as they related to `uintptr_t` arithmetic. The *address* interpretation of capabilities turns this previous compatibility issue into a minor performance deficiency. Additionally, 19 of the changed lines were required to work around the inclusion of the tag bit in comparisons, which is no longer the case.

Overall, this widely used complex program was remarkably easy to port to pure-capability C. Moreover, the porting effort was undertaken before we had improved compiler diagnostics or a working debugger. This indicates that even highly optimized programs can be ported to CHERI with minimal effort.

6.1.3.4 PostgreSQL

PostgreSQL [229] is a widely deployed, enterprise-grade, high-performance relational SQL database. One problem when porting PostgreSQL was that it contains copies of many libc functions, such as `qsort()` or `printf()`, for portability across operating systems. For example, we had to change PostgreSQL to use `qsort_r()` from CheriBSD's libc instead of the local implementation that did not maintain tags because it copies one `long` at a time. In one of the calls to `mmap()` used for shared memory, we had to round up the size due to capability precision (see Section 3.4). A similar issue was found in a call to `shmat()` which we fixed by passing the `SHM_RND` flag to round the size. We also changed some uses of un-prototyped functions to use prototypes (see Section 3.3.1) but only one of these changes has a run-time effect on CHERI. Additionally, we disabled the old-style function interface as it does not use function prototypes and expects integer and pointer values to be in the same register file. Finally, PostgreSQL uses the type `Datum` (which is `uintptr_t`) for most data. Due to this being 128 bits in pure-capability mode, we also had to adjust eleven `#if SIZEOF_DATUM == 8` pre-processor checks to use greater-or-equal instead.

6.1.3.5 libc++

One of the most time-consuming software ports to CheriABI was the C++ standard library libc++. However, this was not caused by fundamental compatibility issues in libc++ but because this was the first C++ project that we compiled in pure-capability mode. Therefore, we discovered many compiler bugs and missing C++ features in CHERI Clang (see Section 3.7). The actual source code changes to libc++ (only 133 SLOC) are mostly cases such as adding new template specializations for the CHERI `__uintcap_t` data type. Additionally, I had to make many changes to the test infrastructure to be able to run it on QEMU.[6] Finally, I also made changes to support CheriSH (see Section 5.4.1.6). Overall, the amount of changes was surprisingly small for such a low-level system library, and we

[6]By default, all tests can be run locally or using QEMU user-mode emulation. However, for CHERI we have to boot CheriBSD and run the compiled binaries remotely.

believe that this would also apply to other C++ standard library implementations such as libstdc++.

6.1.3.6 QtBase library

Qt is a collection of C++ libraries that can be used to write desktop, mobile and embedded applications. Additionally, Qt can be used as the GUI framework for WebKit, and we used it as the foundation of our WebKit port to CHERI (see Section 6.1.3.9). It has been publicly available since 1995 [21] and is widely used. QtBase was the first large C++ library after libc++ that I ported to run as pure-capability code. Therefore, the process was primarily delayed by compiler bugs — which surprisingly were not discovered by compiling and running all libc++ tests. Examples include using `uintptr_t` as the underlying type of bitfields and C++11 strongly-typed enums. The largest change that was required for QtBase was adding support for `uintcap_t` to `QVariant` and the Qt meta-object system. This is required since `uintptr_t` is no longer the same type as `size_t`. QtBase was also the first time I discovered a problem with using the low bits of pointers in comparisons (see Section 3.6.1.2). Additionally, we found the interesting pattern of storing offsets relative to the `this` pointer in `QString` [83] (see Section 3.9.2).

6.1.3.7 SQLite

SQLite [218] is an in-process relational database that is used by many projects, including the FreeBSD package management tool and WebKit. We only had to make two changes (touching 20 SLOC) to SQLite to use it in WebKit. The first problem was that the `SQLITE_INT_TO_PTR` and `SQLITE_PTR_TO_INT` macros were casting via `ptrdiff_t` instead of `uintptr_t`. Using `ptrdiff_t` does not retain the CHERI tag bits or metadata and causes a tag violation at run time. The compiler flagged this issue and it was trivial to fix by replacing `ptrdiff_t` with `uintptr_t`. The second problem was related to the SQLite memory allocator, `sqlite3MemMalloc()`, which wraps the system allocator and stores an additional `uint64_t` at the beginning of the allocation. This causes the return value to not be sufficiently aligned to store CHERI capabilities. The problem was only discovered at run time, but was easily fixed by storing a `uintptr_t` instead of `uint64_t`.

6.1.3.8 rsync

I recently ported the file synchronization utility `rsync` to work as a pure-capability binary. Out of the 41,956 SLOC, only one had to be changed.[7] A variable declaration (`orig_umask`) was incorrectly using `int` (four bytes) instead of the actual type of the definition, `mode_t` (two bytes). Since the pure-capability linkage model (see Chapter 4) tightly bounds all global variables, this deficiency in the C programming language [122,

[7] I also changed two more lines to fix a warning that is an error by default for CHERI, but in this case did not affect program behaviour.

§4.5] resulted in a run-time trap. In this case pure-capability C allowed us to find a real bug that can result in program failures on big-endian systems. We believe that `rsync` is a representative sample for compatibility of small to medium-sized programs and matches our experience with the utilities in the FreeBSD base system.

6.1.3.9 WebKit

WebKit is a framework for rendering web pages and is used for example by Apple's Safari Browser. It is a large codebase with over 2.5 million lines of code and includes full support for JavaScript, CSS and HTML. This was by far the most complicated porting effort of all userspace software we[8] have come across. However, this is to be expected for a complex JavaScript engine that has been highly optimized over the years. One of the biggest problems was the interchangeable use of integers and `uintptr_t`, and the fact that a lot of the code was generated by Ruby scripts [94]. In CHERI-MIPS, pointers and integers live in separate register files and therefore we must be careful to access the appropriate value. Additionally, most of the JavaScript interpreter is written in a 'high-level assembly language' and the actual code (either target-specific assembly or generic C++ code for other architectures) is generated.

In our port of WebKit we use CHERI capabilities for all JavaScript object references instead of offsets into a JavaScript heap. This allows us to check the tag bit to differentiate objects and floating-point values, thus eliminating all vulnerabilities that rely on type confusion between `double` and `JSObject` such as CVE-2016-4622 [89].[9]

The biggest challenge here was not CHERI compatibility (1661 SLOC) but understanding the complex codebase and dealing with compiler bugs. Importantly, we completed this port in a few months — without any prior knowledge of the codebase — and only working on this task part-time.

6.1.4 Detailed case study: LLVM libFuzzer runtime

As a final case study, I ported the LLVM runtime libraries (compiler-rt) for sanitizers such as UBSan, ASan and the libFuzzer fuzzing tool [204]. Most of the porting so far was done with only a subset of the new compiler warnings and therefore involved a lot of manual debugging. To evaluate the applicability and helpfulness of the warnings I ported a new piece of software to CHERI. I chose this library since it would almost certainly be a worst-case[10] low-level userspace library in terms of porting effort. It turns out that this project contains many pieces of code that have proven to be problematic in the past:

- It contains a custom allocator so that it does not need to depend on libc's `malloc()`.

[8] I ported QtWebKit's dependencies such as QtBase, ICU4C and libxml, while Khilan Gudka ported the JavaScript language runtime and other aspects of WebKit.

[9] However, we still use the existing *NaN-boxing* [93] representation for non-reference types.

[10] Excluding operating-system kernels, base system libraries and managed language runtimes.

- Most of the libc functions — even `memcpy()` — are re-implemented in the sanitizer runtime. This is done so that the sanitizers (e.g. UBSan) can also work in free-standing environments such as operating-system kernels.
- It uses `uintptr_t` extensively but has an incorrect definition. It also uses `uintptr_t` in places where a `size_t` should have been used instead.
- It implements its own version of `printf()`.

Issues found at compile time In the processes of porting the LLVM sanitizer runtime to CheriABI, most necessary changes were indicated by the new compiler warnings:

- I found many casts between pointers and non-`uintptr_t` integers, which were the result of using (`unsigned`) `long` for the `uptr`/`sptr` internal `uintptr_t` type alias. I fixed this by changing them to use `__(U)INTPTR_TYPE__`, which is the correct type on all architectures.
- After fixing this, the compiler warned about many instances of `uintptr_t` arithmetic. After the introduction of the virtual-address-based capability interpretation (see Section 3.6) these are no longer real problems but do highlight inefficiencies: using 64-bit arithmetic is both faster[11] and allows reducing the size of data structures.
- The compiler warned about subtractions between `uintptr_t` types as this can cause errors in the *offset* interpretation of CHERI. Although this is not an issue in *address* mode, it does make the code slightly less efficient and this warning highlights variables that should be integers rather than `uintptr_t`.
- Finally, the CMake code checked that `sizeof(void*)` was 4 or 8. This check is not required after my fixes, so I removed it.

Issues found at run time After fixing the compiler warnings, I discovered seven (small) issues that were not detected statically:

- A function pointer obtained from `dlsym()` was cast to a virtual address and stored in a `uintptr_t` variable. This untagged value was called later, causing a crash.
- The runtime contained `internal_memcpy()` and `internal_memmove()` functions which did not retain tags. They are used so that UBSan/ASan can be used in environments that do not provide a full C library, but a byte-by-byte copy does not work for CHERI as it strips the capability tags (see Section 3.9.4).
- The libFuzzer runtime depends on intercepting UNIX signals but did not provide one for `SIGTRAP` (raised by `__builtin_trap()` on MIPS) and `SIGPROT` (a new signal specific to CheriBSD that is raised on a capability error). This problem cannot be detected by static analysis but was obvious when running the program.
- There was a copy of uninitialized memory from the stack. This was detected because `memcpy()` by default warns about stores of capabilities to under-aligned locations

[11] CHERI-MIPS, being a split register-file architecture, requires additional `CGetAddr`/`CSetAddr` instructions.

(see Section 3.2.4). This was fixed by invoking `memset()` before the call to the copy constructor. We might not have noticed this when compiling with optimizations enabled since the copy from the stack could be elided.

- The internal allocator only aligned allocations to 8 bytes. This is insufficient for storing a capability and was discovered due to a run-time crash when using libFuzzer.
- The runtime also implemented its own version of `printf()` which loaded a `uintptr_t` if the `%z` modifier was passed. This is broken for CheriABI, since `va_arg()` fetches 16 bytes from the stack instead of the intended 8 bytes for `size_t` (see Section 3.3.1.2 and [53]). Normally, this would only be found at run time, but I discovered it while looking at the `printf()` code, and we know from prior experience that `printf()` is often not implemented in a CheriABI-compatible way.
- Furthermore, the codebase used the macro `__LP64__` to determine the integer register size and assumed 32-bit integers since this macro is not defined for pure-capability C++. This was only discovered by chance when I added an assertion to check that the internal `ptrdiff_t typedef` matches the compiler provided `__PTRDIFF_TYPE__`.

The misuse of `dlsym()` could have been detected by adding a very noisy warning: we could warn every time a conversion between `(u)intcap_t` and integer types happens without an explicit cast. This warning would require conversions to be audited, but unfortunately many already have an explicit cast. Therefore, we would need an explicit `__cheri_addr` cast to silence the warning. However, we believe that this warning has a low signal-to-noise-ratio, so we have not yet implemented it. The `internal_memcpy()` problem could be detected by a static analyser or the compiler (which already replaces such loops with `memcpy()` if they match certain patterns). However, the code was explicitly written so that the compiler patterns do not match (and is also compiled with `-fno-builtin`), so the compiler cannot perform this transformation.

Bugs found CHERI is a memory-safety enforcement and vulnerability-mitigation tool, not a debugging tool, so we do not generally expect to detect many bugs. These bugs have ideally already been found with more suitable (and less well-performing) tools, and yet we do still find some bugs due to the feature set of CHERI.

I discovered an out-of-bounds access in libFuzzer that we believe can (currently) only be detected using CHERI. The function `ForEachNonZeroByte` includes a loop that reads eight bytes at a time until the end of the buffer. However, the loop condition was wrong and therefore could read up to seven bytes past the end of the buffer if the size is not a multiple of eight (which was the case in my test binary).

Tools such as ASan or Valgrind will detect this error for heap buffers. However, the buffer used here is a compiler-generated buffer for the fuzzing instrumentation, `__start___sancov_cntrs`. As the variable is a data symbol that is provided by the static linker, most other tools do not have the required bounds information. However, for CheriABI I modified LLD to always emit `__start_<section_name>` symbols with

bounds matching the size of the ELF section. Therefore, we were able to catch this kind of overflow that other tools cannot detect.

Case study applicability This case study highlights the worst-case amount of changes as libFuzzer is a very low-level library, yet even so the number of changed SLOC was only 1.81%. The majority of issues were discovered through compiler warnings and should be trivially fixable by following suggestions from the compiler. The other issues were mostly related to libFuzzer using its own version of libc functions and should therefore not occur in most software.

This is not a well-formed study, in that a single (experienced) CHERI software developer did the work, but anecdotally it offers interesting insight. We believe that source-code modifications are a more objective measure of compatibility.

6.1.5 Summary

CHERI pure-capability compilation requires very few changes: around 0.01–0.14% of SLOC in most cases.[12] This is significantly lower than the 2.4–4.5% of SLOC measured previously for hybrid CHERI (annotating pointers with `__capability`) [41]. Interestingly, many of the pure-capability changes address implementation-defined/undefined behaviour in C/C++ that happen to work on contemporary architectures.

Overall, the porting effort is much lower than in any other previously published memory safe C implementation. For example, '*Cyclone*' requires around 10% of all lines to be modified [116] according to the authors' estimates. Similarly, '*Checked C*' requires changes to 9.8–35.3% of all lines [67]. These numbers have recently been improved upon by using a Clang-based tool to automatically translate C to Checked-C [193]. This tool can convert 23–46% of all language-visible pointers to checked variants, but the remaining cases still need to be investigated manually. I was unable to find overall change percentages for '*CCured*' [162] but Necula et al. report 2000 changed lines for OpenSSL (CHERI requires 42) and 365 for OpenSSH (CHERI does not require any).

6.1.6 Future work

We have identified many patterns that cause compatibility issues for pure-capability C/C++ (see Chapter 3) and added compiler diagnostics to identify some of them at compile time. However, there are still many more diagnostics that I would have added if I had more time.[13]

[12] For most projects this also includes the changes required by CheriSH to enforce sub-object protection.

[13] Besides working on the compiler, I also improved QEMU, LLD, CheriBSD, ported third-party libraries such as Qt to CHERI, and maintain our Jenkins continuous integration setup.

6.2 Performance

While security benefits have been becoming more important recently, it is still not considered acceptable to ship such a feature if it results in noticeable performance losses [221]. Therefore, it is paramount to show that CHERI pure-capability code has low overheads compared to an insecure baseline. We evaluate this performance difference using various well-known benchmark suites.

Unless stated otherwise, all benchmarks were executed 10 times. Graphs show the median overhead with error bars highlighting the inter-quartile range. We measured the cycle count (i.e. total benchmark duration), instructions executed and L2-cache misses. Presenting instruction count is useful for two reasons: our micro-architecture is sensitive to instruction bloat and additionally instruction count overheads (or improvements) indicate compiler code-generation deficiencies. We also report L2-cache misses as these result in DRAM accesses[14] and are therefore indicative of power usage and performance overheads caused by increased data structure size.

6.2.1 Evaluation platform

For micro-architectural realism, all benchmarks in this section were run on a version of CHERI on a Stratix IV FPGA at 100MHz. The pipeline is in-order and single-issue, roughly similar to the ARM7TDMI. Our FPGA system has 32KiB L1I/L1D caches and a shared 256KiB L2 cache, all set-associative, like widely shipped CPUs such as many ARM Cortex A53 implementations, although without pre-fetching. Performance and memory scaling are broadly similar to these commercial implementations. Specific performance is subject to the peculiarities of our microarchitecture. We ran all benchmarks on CheriBSD rather than in a bare-metal environment, so all measurements also include virtual memory and context switching costs.

6.2.2 Benchmarking challenges with CHERI and MIPS

We have attempted to reduce any differences between MIPS and pure-capability CHERI code to a minimum to ensure that this performance evaluation measures the impact of replacing integer pointers with CHERI capabilities without being affected by unrelated issues. Moreover, we compile MIPS code without any additional security mitigations such as stack-protector and run all benchmarks without ASLR.

Benchmarks for CheriABI and MIPS are both compiled with the CHERI compiler (based on LLVM 9) and use the CheriABI kernel. For pure-capability code we use the PC-relative ABI (see Section 4.2.4) as this closely models the instruction sequences used by the MIPS n64 ABI.

[14] This is true on our simple in-order pipeline but may not be for more complicated processors.

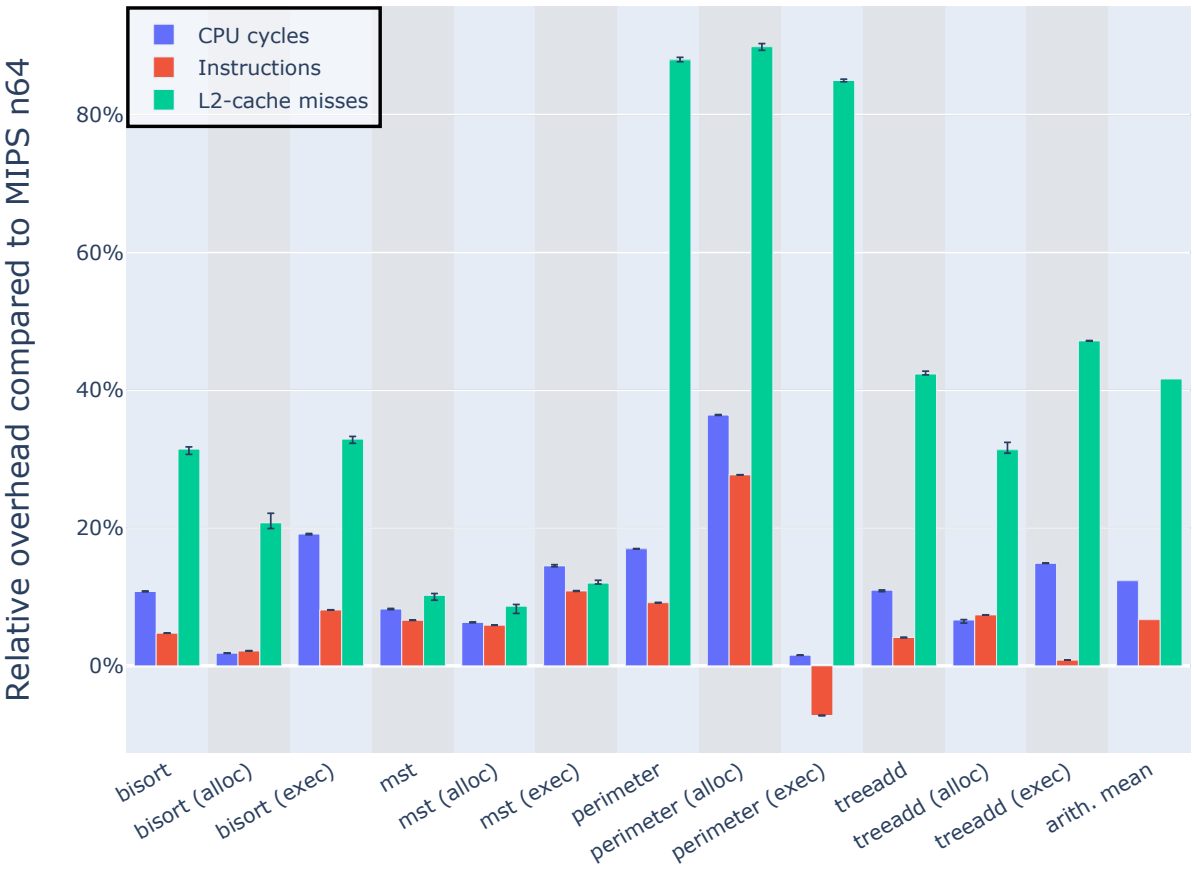


Figure 6.1: Olden pure-capability performance relative to MIPS baseline.

We ensure that the more efficient `memcpy()` (copying data with capability registers) is used in both cases. Furthermore, we compiled all MIPS binaries in the hybrid CHERI compilation mode, which allows the compiler to inline `memcpy()` with capability-sized copies. This allows MIPS code to copy two pointers at a time. One performance disadvantage for pure-capability code is that it cannot inline `memcpy()` for sizes greater than a capability if the allocation is not capability-aligned (see Section 3.2.4).

However, there are some unavoidable architectural differences caused by the split register file (see Section 6.2.7). Additionally, CHERI-MIPS uses differently encoded memory access instructions that have a smaller immediate range (which negatively affects some benchmarks), but also provide a register offset operand (which has a positive effect on other benchmarks). For our upcoming RISC-V version of CHERI we will explore both split and merged register files as part of the design parameter space and use equally expressive memory access instructions. However, CHERI-RISC-V was not ready to be used at the time this research was performed.

6.2.3 Olden

First, we look at the Olden benchmark suite [192] which is a very pointer-intensive benchmark suite (originally developed at Princeton for their Olden runtime) and should

therefore be hit the hardest by doubling the pointer size. The benchmark suite has also been used for evaluation in other memory safety work [58, 59, 60, 160, 260]. Due to the CHERI FPGA being built without a floating-point unit (FPU), we omit the floating-point benchmarks in this suite. We ran all benchmarks 15 times. All Olden benchmarks include an initial allocation phase where the input data structure is created and an execution phase where the actual computation is performed. Due to the volatility of jemalloc [72] performance depending on allocation sizes, we modified the benchmarks to also report performance numbers after each phase.

The use of pointer-dense data structures can be seen by the higher number of L2-cache misses compared to the MIPS baseline (see Figure 6.1). As pointers are twice the size with CHERI, greater use of pointers results in a higher number of cache misses. This is especially noticeable in the perimeter benchmark which has an L2-cache overhead of 88.0%, i.e. the data footprint has almost doubled. In this benchmark the main data structure contains five pointers and two integers. Therefore, the data footprint doubles from 48 bytes to 96 bytes (the CHERI layout adds 8 additional bytes due to padding for alignment). However, even for the worst-case, the allocation phase of perimeter, we require only 36.4% more cycles than the MIPS baseline even though we are allocating twice as much memory. This overhead might be alleviated by using a different memory allocator that is not as micro-optimized for specific workloads as jemalloc. If we look at the entire benchmark run, we see overheads of 10.8% (bisort), 8.2% (mst), 17.0% (perimeter) and 11.0% (treeadd). Olden has been useful for identifying compiler code-generation problems. For example, the execution phase of treeadd originally had over 40% instruction count overhead due to missed compiler optimizations.

6.2.4 MiBench

We also evaluate performance against '*MiBench*' [97], a benchmark suite originally designed to be representative for embedded workloads. Unlike Olden, these benchmarks are not pointer intensive, and we can therefore expect better performance. As can be seen in Figure 6.2 only four of the benchmarks have a significantly higher number of L2-cache misses, which shows that pointer-size increase is not a limiting factor.

The pure-capability code performs very well on this set of benchmarks, with many benchmarks using fewer cycles than MIPS and only four executing more instructions. CHERI-MIPS has a split register file so many benchmarks can make use of the additional callee-saved registers and therefore perform better (see Section 6.2.7). The security-sha benchmark performs especially well on CHERI because the added capability registers reduce the number of stack spills in the main benchmark loop. This performance win due to reduced register pressure would almost certainly be lower if we were able to run the benchmarks on an architecture that includes instructions for vectorizing loops or accelerating cryptographic operations.

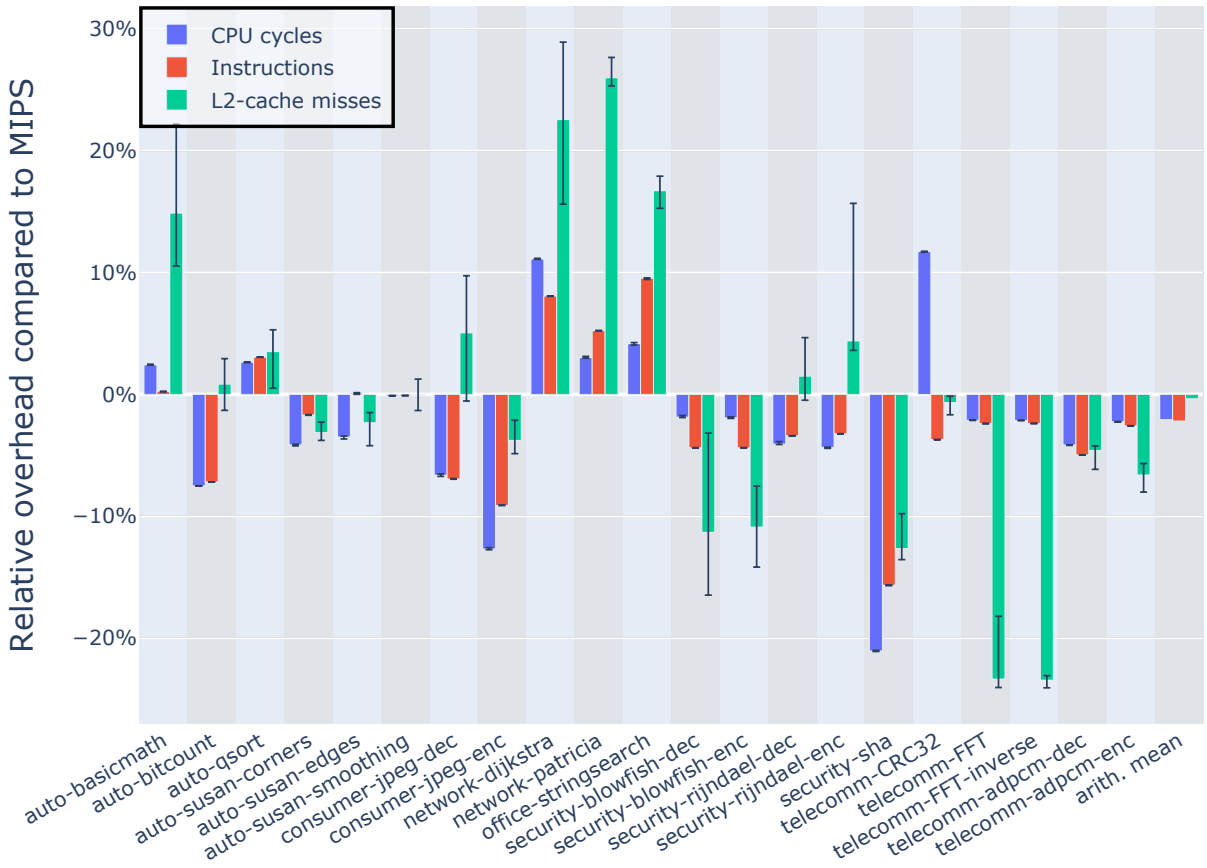


Figure 6.2: MiBench pure-capability performance relative to MIPS baseline.

6.2.5 SPEC CPU 2006

Finally, we also run a subset of the SPEC2006 benchmark suite [100], more specifically the integer part of SPEC 2006 [220]. In this case we ran 10 iterations of each benchmark using the test dataset, since using the ref dataset requires up to 2GB of memory [101, 219] and our FPGA only has around 500MiB available. We do not attempt to run the floating pointer benchmarks as we build our FPGA without a FPU by default.[15] Figure 6.3 shows that five out of nine SPEC2006 benchmarks execute faster as pure-capability binaries.[16] As with MiBench, see Section 6.2.7 for an explanation of this effect. The only benchmarks that run noticeably slower for CHERI are 471.omnetpp and 483.xalancbmk. Yet even for these worst-case benchmarks we incur overheads of only 14.5% and 23.3% respectively. It turns out these benchmarks spend a much their execution inside `malloc()` and `free()`, and the current pure-capability implementation of `free()` requires additional logic compared to the MIPS baseline. Additionally, 483.xalancbmk uses pointer-intensive XML data structures, so we see more frequent L2-cache misses. For 471.omnetpp, much of the slowdown is caused

[15] A configuration with a FPU is available. However, by default all system libraries are built with software floating-point, so we would not be able to link against system libraries.

[16] Of the 12 SPEC benchmarks, we do not run the 400.perlbench, 403.gcc and 429.mcf benchmarks since these programs contain various CHERI-incompatible patterns (e.g. using high pointer bits and incorrectly relocating pointers after `realloc()`) and the SPEC build system makes modification of sources difficult.

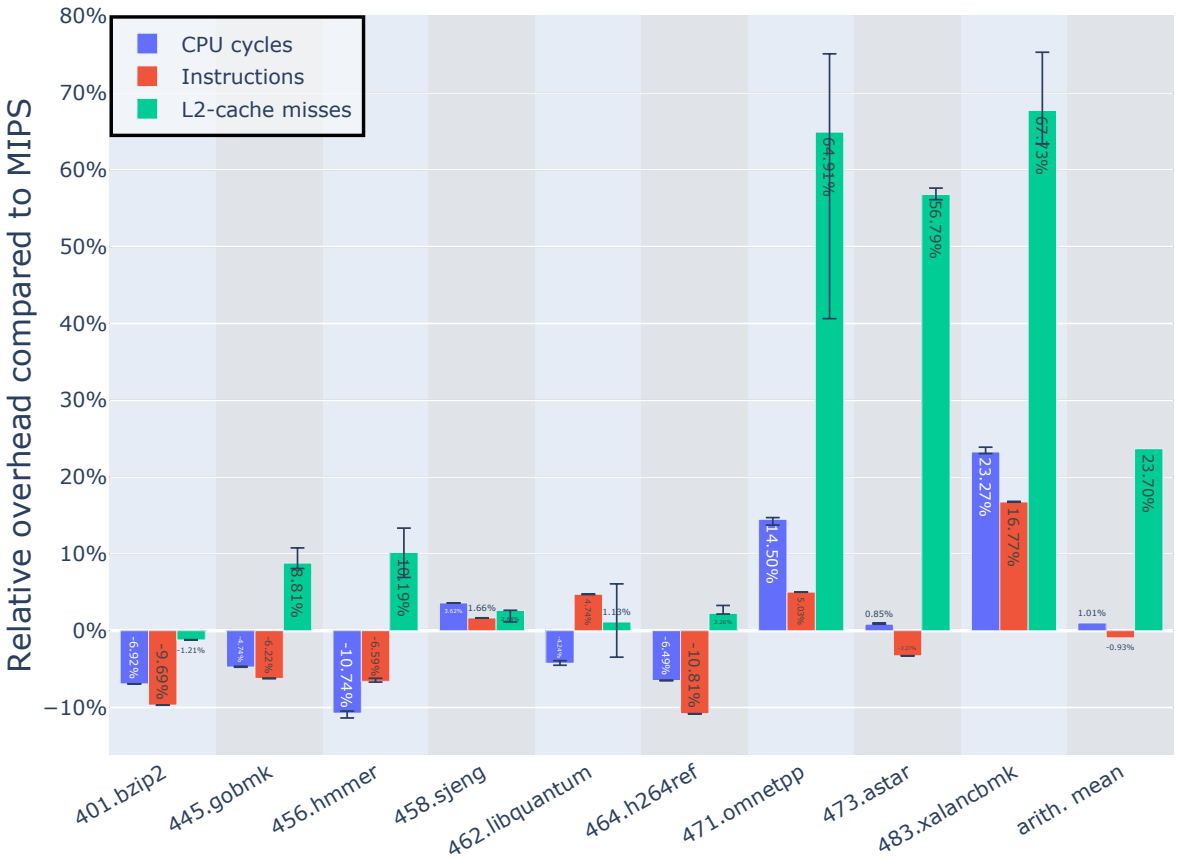


Figure 6.3: SPEC CPU 2006 pure-capability performance relative to MIPS baseline.

by increased TLB misses.[17] In the MIPS case, the live working set almost fits within the 2MB covered by the TLB, whereas for pure-capability CHERI we cross this threshold and therefore incur 44 times as many TLB misses.

For this series of benchmarks, pure-capability code runs slightly slower on average (1.0% arithmetic mean, 0.5% geometric mean). However, the benchmark that runs slowest incurs high instruction-count overheads in addition to higher L2-cache misses. This indicates that it suffers from poor compiler code-generation that could be improved in the future.

6.2.6 PostgreSQL

As a large real-world benchmark, we also tested the PostgreSQL relational database. The first benchmark we ran was the `initdb` program [230]. This program sets up a new SQL database and while doing so it forks multiple child processes and uses inter-process communication (IPC) to communicate with them. Therefore, this program was also a good test case to check that the IPC system calls[18] work correctly with the CheriABI system call layer. We measured the whole-system performance counters from starting `initdb` until it terminates. As can be seen in Figure 6.4, this process required 5.7% more

[17] Due to MIPS having a software-loaded TLB, at least five of the 14.5% cycle overhead and almost the entire instruction overhead can be attributed to the TLB miss handler.

[18] This includes the System V shared memory calls such as `shmget()` and `shmctl()`.

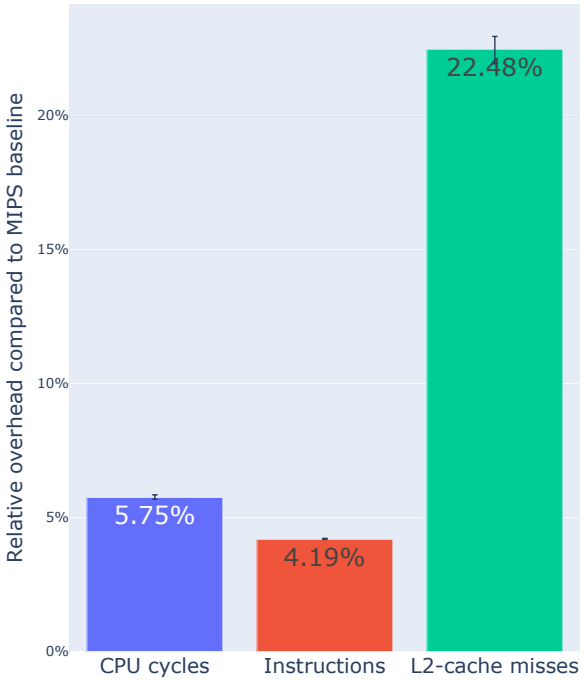


Figure 6.4: Performance for pure-capability PostgreSQL `initdb` relative to MIPS n64.

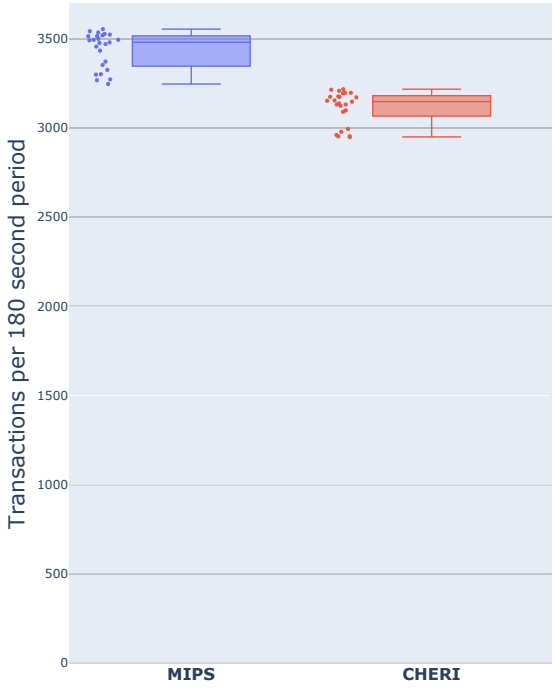


Figure 6.5: Completed pgbench transactions in 180 seconds (25 samples).

cycles as a pure-capability binary, despite data structures not being optimized for CHERI. In contrast to this, a initdb binary with ASan instrumentation (but *without* instrumented library dependencies) requires 229% more cycles to complete [54].

The second benchmark is the pgbench benchmark utility [231] that is shipped as part of the PostgreSQL source tree. This benchmark runs a fixed sequence of SQL commands and reports the number of transactions completed per second. We ran this benchmark 25 times with the following configuration: one thread, one client, and 180 seconds duration. As we ran the benchmarks on a single-core CHERI FPGA, it does not make any sense to use more than one thread. Figure 6.5 shows that the median number of transactions completed by the CHERI pure-capability binary is 9.6% lower than the MIPS n64 baseline (3148 transactions instead of 3481). We believe that the number of system calls performed by the benchmark is one reason that this number is higher than for initdb. CheriABI is currently implemented as a compatibility ABI in the CheriBSD kernel, which results in additional overhead for translating structures to call kernel-native functions. Additionally, we did not attempt to avoid gratuitous padding inside PostgreSQL data structures or replace uses of `uintptr_t` with the appropriate integer type. We believe that this overhead can be reduced by improving compiler optimizations. Furthermore, MIPS n64 benefits from the faster `memcpy()` implementation and the ability to inline copies of some data structures that are not inlined for CHERI.

6.2.7 Performance-effects of a split register file

Some benchmarks have a lower cycle count compared to MIPS n64 even though the number of instructions is higher. This is a side effect of CHERI using a separate register file for capabilities, i.e. having 32 new capability registers in addition to the existing 32 integer registers. The separate register file adds eight callee-save registers that can be used to retain values across calls. This makes a significant difference as LLVM is quite aggressive when it comes to inlining, so many performance-critical functions end up preserving more values across calls than can be kept in registers. When analysing the benchmarks with this effect, it turned out that some dense loops include reloads from the stack in the MIPS case, whereas the pure-capability CHERI code can move from a register instead. Moreover, for most of these reloads, the compiler uses the loaded value in the next instruction. On the pipelined CHERI FPGA this results in an additional two cycle delay until the value is usable by the next instruction.

These performance advantages for CHERI would be less pronounced for architectures with a merged register file, i.e. with general purpose registers extended to hold capabilities.[19] However, a merged register file implementation would also have performance advantages due to not having to save an additional 32 registers per context switch. Furthermore, if we had a merged register file, this would also remove significant overheads related to moving values between capability and integer registers as the integer operation could be used directly (see Section 3.8.1). Therefore, it is highly benchmark-dependent whether this would result in improved or reduced performance. Nevertheless, we believe that for future implementations of CHERI, a merged register file is the better choice as it can simplify the toolchain and improves C compatibility regarding calling conventions (see Section 3.3.1).

6.2.8 Performance improvements since prior publications

The performance numbers reported in this section differ substantially from the results published earlier this year [53, 54, 254], as pure-capability code generation has improved noticeably since then. I added new compiler optimizations such as teaching LLVM about CHERI intrinsics in some optimization passes. Furthermore, I introduced new instructions (see Section 3.8.1) which can noticeably improve code density. The largest performance win so far was omitting bounds that can be shown to be unnecessary at compile time (see Section 3.8.2). Before this optimization, we would unnecessarily save many registers in most functions. This negated the performance win from having more callee-save registers available (see Section 6.2.7). Some passes in LLVM have been disabled or are less efficient for CHERI and fixing them to handle CHERI capabilities would require significant work.

[19] It is unclear whether all 32 register should be extended or only a subset of them. Most functions do not need 32 pointers so not widening all registers could reduce power and space consumption.

We therefore believe that further performance improvements compared to the baseline are still possible.

6.2.9 Performance of other spatial safety tools

One of the most commonly used memory-safety tools is AddressSanitizer (ASan) [201]. SPEC2006 performance for ASan is reported to have 69% CPU and 237% memory overhead [201]. However, more recent publications report the CPU overhead to be 80% [128] or 99% [216]. A hardware-assisted version of ASan (HWASAN) leveraging ARM top-byte-ignore exists. This reduces memory overhead to 10–35% [9] but still incurs similar CPU overheads [205, 206]. Ideally, we would compare our performance to the future HWASAN version based on memory tagging, but that does not exist yet [202]. Moreover, ASan is a probabilistic technique by design (even with memory tagging it has a 1/16 chance of failure) rather than a deterministic vulnerability-mitigation tool. Since ASan's overheads are too large to be used as a bug detection system in production, Google only runs a small subset of its servers with ASan enabled [205].

Despite being hardware-accelerated, Intel MPX is no faster than ASan. Oleksenko et al. measured the overheads of these spatial-safety mechanisms [174, 175] and reported an *average* overhead of 46% (with the Intel compiler) or 139% (with GCC). Similar hardware bounds-checking mechanisms include '*Hardbound*' [58] (2–22% overhead for Olden), '*Watchdog*' [157] (24% average and ≈50% worst-case overhead for SPEC2006). '*WatchdogLite*' [158] (29% average and ≈100% worst-case overhead for SPEC2006). However, like MPX they use out-of-band bounds metadata so are unlikely to be much faster than MPX in practice.

'*SGXBOUNDS*' and '*Delta Pointers*' provide bounds checking with 64-bit pointers but restrict the address space to 32-bits. The reported SPEC2006 overheads for '*SGXBOUNDS*' are 41% overhead inside enclaves and 55% outside [129] (or 94% according to [128]) and 35% for '*Delta Pointers*' [128]. The '*Low-Fat Pointers*' hardware-enforced bounds checking technique uses only 18 out of 64 bits for bounds metadata and claims to have no execution overhead (with 3% memory overhead) compared to 64-bit pointers [132], but it cannot enforce byte-granular bounds. A software implementation inspired by this technique incurs 64% overhead for stack and heap protection [64, 65].

Approaches that require modifications to source code can result in lower overheads. '*Checked C*' reports a run time overhead of 0–49.3% (mean 8.6%) [67] but incurs high porting efforts and '*CCured*' [162, 163] reports 3–87% overhead. The numbers reported for '*Cyclone*' are 9%–185% overhead for a selection of six benchmarks [116].

Performance overheads for some other bounds checking techniques include the '*Valgrind*' MemCheck tool [166] with 1960% overhead for SPEC2006 [216], '*SoftBound*' with 67% overhead across SPEC2006 and Olden [160] and '*MemSafe*' with 15–205% overhead for various benchmarks including Olden (15–55%) and SPEC2006 (120–183%) [209]. Tools

evaluated using SPEC2000 report 72% overhead for '*Baggy Bounds Checking*' [6], 49% for '*PAriCheck*' [263] and 23% for '*Light-Weight Bounds Checking*' [99]. The *Memory Safe C Compiler* (MSCC) incurs 133% overhead for Olden and 146% on SPEC2000 [260].

Importantly, the overheads reported in these papers are mostly *geometric* means, a metric that causes outliers to have little effect on the overall results. However, for memory-safety (almost) all outliers will be overheads rather than speedups and these outliers are generally the benchmark programs affected most by the memory-safety techique. The benchmark suites used to collect the average overheads all contain multiple programs that are either I/O-bound or purely computational and therefore do not measure the overheads of memory safety. Including such benchmarks artificially reduces the geometric mean and — while common practice in many publications — using such a number as the overall result can be misleading.

While the geometric mean is often considered the correct metric for summarizing benchmark results [75], 'any measure of the mean value of data is misleading when there is large variance' [75] and most of these results have a very large variance. For memory safety techniques, the most meaningful metric is the worst-case overhead since any mean overhead (arithmetic, harmonic or geometric) can be reduced by adding more low-overhead benchmarks.

6.2.10 Summary

Overall, CHERI pure-capability code incurs significantly lower overheads (and provides stronger protection guarantees) than any other spatial memory protection scheme. This includes software-only schemes (e.g. ASan [201] or '*SoftBound*' [160]) as well as hardware enforced models such as Intel MPX [174].

If we look at cycle overhead (i.e. absolute run time) of pure-capability code, the *worst-case* overhead is only 23.3% for the very-pointer heavy synthetic benchmark 483.xalancbmk. This worst-case overhead is better than the average (or even best-case) overheads for most prior techniques (see Section 6.2.9). On average (arithmetic mean), pure-capability code runs only 0.5% slower than MIPS. If we use a geometric mean, which is commonly reported in memory-safety publications, we see an even smaller number of 0.1%. This result is unintuitive — considering the doubling of pointer size for CHERI pure-capability code — but for many benchmarks pointer size is not a limiting factor. Admittedly, some benchmarks benefit from the larger split register file (see Section 6.2.7) and there would probably be fewer benchmarks that run faster in pure-capability mode on a merged-register-file architecture. However, this does show that for the MIPS ABI, the added callee-save registers provided by CHERI have a greater impact than the increased pointer size. Even if we look only at the benchmarks that are slower than MIPS, the geometric mean overhead is 8.5%. All this indicates that ABI design choices could actually be more important for performance (especially with aggressively inlining compilers) than the size of pointers.

If we look at a real-world workload, we can see that CHERI pure-capability PostgreSQL (see Section 6.2.6) takes 5.7% longer to create a database. This indicates that extending ISAs with CHERI and compiling all code with pointers implemented as capabilities results in a performance overhead below the limit of 5–10% overhead for real-world deployment [221]. On-by-default security mitigations such as stack-protector [50], some CFI implementations [28] and speculative execution vulnerability mitigations can come at a higher cost. For example, running the LLVM regression tests with KPTI takes 4 times longer on a virtualized FreeBSD build server but this mitigation technique is deployed by default [91]. Many of these mitigations are no longer necessary with CHERI's complete spatial safety,[20] so it is possible that deploying CHERI could in fact improve performance for most code.

6.2.11 Benchmark availability

A recent version of the Olden and MiBench benchmark suites can be found as part of the LLVM test suite [142]. SPEC CPU 2006 is available for purchase from SPEC at https://www.spec.org/order.html. The PostgreSQL benchmarks were performed using a slightly modified (see Section 6.1.3.4) version of PostgreSQL 9.6 [48].

[20] Additionally, architectural support for compartment boundaries (`CSetCID` [246, 249]) could significantly reduce the cost of speculative execution mitigations.

7 CONCLUSION

In this dissertation I have presented challenges and refinements to pure-capability C/C++ semantics (see Chapter 3), protection strategies for language-invisible pointers (see Chapter 4) and sub-objects (see Chapter 5) and shown that pure-capability C/C++ is a highly compatible and performant programming model (see Chapter 6). Pure-capability CHERI compilation can improve inherently unsafe languages such as C and C++ in many ways. These improvements focus on (but are not limited to) the following aspects:

Highly compatible and performant memory safety Memory buffer handling errors were declared the most dangerous software error of 2019 [45], which highlights the importance of using techniques such as CHERI that can provide memory safety and thereby deterministic rather than probabilistic defences against vulnerabilities. Pure-capability C/C++ retains almost complete source-level compatibility with existing code: in many cases recompiling with the CHERI LLVM compiler will yield a fully functional binary. In cases where this is not true, I have shown that usually fewer than 0.1% of lines require modifications. Therefore, the barrier for pure-capability C/C++ adoption should be low, especially after having refined the semantics to improve compatibility with existing code (see Chapter 3). Moreover, CHERI is an extension to existing ISAs and allows for incremental adoption: programs with source code available can be recompiled to attain memory safety and existing binaries will run as before on CHERI-enabled hardware.

In terms of performance, I have shown that CHERI pure-capability code has remarkably similar performance to a 64-bit integer pointer MIPS baseline, and in some cases can even be faster. Existing security mitigation features with worse performance characteristics (and *fewer* security benefits) than CHERI have been deployed in the past (e.g. some CFI mechanisms or speculative execution mitigations). The 23.3% worst-case and much lower geometric mean overhead of 0.1% for CHERI pure-capability code is vastly better than prior memory-safety techniques and should be low enough for real-world deployment [221].

Defence against tomorrow's security vulnerabilities Most current exploits rely on data vs. pointer type confusion (mitigated by CHERI's architecturally enforced referential safety) or inter-objects spatial memory violation (mitigated by pure-capability C/C++). Similar to the prevalence of stack canaries shifting exploits from sequential write (e.g. `strcpy()`/`memcpy()`) to targeted out-of-bounds write exploits (e.g. missing array bounds checks) [153], we envision that a real-world deployment of CHERI may lead to these exploit primitives being used less and shift attackers towards techniques more difficult

to successfully exploit such as sub-object data-only attacks (e.g. overwriting adjacent structure members to pass malicious arguments to functions) or temporal safety violations (e.g. reinterpreting an object of an unprivileged type as being privileged). These flaws are already exploited in some cases, but this is quite rare due to the plethora of easily exploitable spatial memory flaws in current software.

While not enabled by default in the pure-capability compilation model, the CHERI architecture provides the foundations for defence techniques against these more sophisticated attackers. This dissertation has presented CheriSH, which can defend against many sub-object-based attacks by providing fine-grained *complete* spatial safety on top of the CHERI pure-capability model (see Chapter 5). Additionally, CHERI's architectural features such as tagged memory can be leveraged to provide temporal safety at much lower overheads than would be possible on a conventional architecture. Originally, we envisioned that copying garbage-collection on top of CHERI would be used to combat temporal violation and had designed the pure-capability C/C++ semantics to accommodate for this. However, this choice resulted in many source-level incompatibilities, so we have since refined the semantics to de-emphasize garbage collection and focus instead on compatibility (see Chapter 3) and explore new promising techniques to combat temporal safety violations such as sweeping revocation [74, 258].

As opposed to many other hardware security features that have recently been introduced (e.g. ARM Branch Target Indicators or Intel Control-flow Enforcement Technology), CHERI is not a technique to mitigate a single exploit class, but rather a foundational building block for architectural primitives that deterministically rule out entire classes of vulnerabilities—and more importantly enables new software designs that were previously impossible.

Future opportunities for compartmentalization and temporal memory safety Another key potential of CHERI is the secure compartmentalization and domain transition mechanism [170, 247, 248]. Domain transitions using existing process- or MMU-based schemes are currently very expensive and are therefore only used in limited, high-risk domains such as web browsers [18, 182]. Recently, Google Chrome increased the level of compartmentalization, now also isolating different sites within the same page to mitigate information leakage due to speculative execution attacks. This increases memory usage by 10–13% and CPU usage by at least 8.2% yet is enabled by default [183] and other browsers are likely to follow this approach [135]. Therefore, we envision that CHERI will *improve* performance if it is used to replace existing vulnerability-mitigation and compartmentalization approaches.

Moreover, CHERI enables many new interesting compartmentalized programming models due to the massively reduced cost of domain transition. In this dissertation, I have extended the CHERI ISA with the new architectural feature of *sentry* capabilities (see Section 4.7.1). This avoids the complexity of object type allocation and revocation that CHERI compartmentalization previously incurred and should simplify the adoption

of compartmentalized software models. Building upon the new linkage models presented in this dissertation, much finer-grained and potentially automatically (or tool-assisted) generation of compartments will be possible in the future.

The pure-capability programming model presented in this dissertation is also an essential foundation for CHERI-based temporal safety. Using CHERI capabilities for all language-visible pointers enables precise tracking of all valid objects and can facilitate techniques such as sweeping revocation [68, 74, 258].

Real-world adoption of CHERI I am very excited by the possibility of future real-world adoption of CHERI. Through its *Digital Security by Design* industrial strategy challenge, the UK government (together with private companies) is providing almost £190 million worth of funding [55] to create a hardware prototype of a mainstream CPU architecture augmented with CHERI [107]. As part of this initiative, 'a recent superscalar ARM Cortex-A class multicore processor will be enabled with Capability Hardware' [234]: the ARM Morello prototype board [13, 87]. This hardware platform will enable exciting future research and potentially even pave the way for the inclusion of CHERI in future generations of ARM CPUs.

I hope that the research and software artifacts that are the basis of this dissertation (and potentially even the document itself) will be a useful and valuable contribution towards a real-world deployment of CHERI.

Bibliography

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson and Jay Ligatti. 'Control-Flow Integrity'. In: *Proceedings of the 12th ACM Conference on Computer and Communications Security* (Alexandria, VA, USA). New York, NY, USA: ACM, 2005, pp. 340–353. DOI: 10.1145/1102120.1102165 (cit. on pp. 16, 27).

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson and Jay Ligatti. 'Control-Flow Integrity Principles, Implementations, and Applications'. In: *ACM Transactions on Information and System Security* 13.1 (1st Oct. 2009), pp. 1–40. DOI: 10.1145/1609956.1609960 (cit. on p. 27).

[3] Reto Achermann, Chris Dalton, Paolo Faraboschi, Moritz Hoffmann, Dejan Milojicic, Geoffrey Ndu, Alexander Richardson, Timothy Roscoe, Adrian L. Shaw and Robert N. M. Watson. 'Separating Translation from Protection in Address Spaces with Dynamic Remapping'. In: *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (Whistler, BC, Canada). New York, NY, USA: ACM, 2017, pp. 118–124. DOI: 10.1145/3102980.3103000 (cit. on p. 21).

[4] Misiker Tadesse Aga and Todd Austin. 'Smokestack: Thwarting DOP Attacks with Runtime Stack Layout Randomization'. In: *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA). Piscataway, NJ, USA: IEEE, 2019, pp. 26–36. ISBN: 978-1-72811-436-1. URL: http://dl.acm.org/citation.cfm?id=3314872.3314879 (cit. on p. 28).

[5] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa and Miguel Castro. 'Preventing Memory Error Exploits with WIT'. In: *Proceedings of the 2008 IEEE Symposium on Security and Privacy* (Oakland, CA, USA). Washington, DC, USA: IEEE Computer Society, 18th–21st May 2008, pp. 263–277. DOI: 10.1109/SP.2008.30 (cit. on p. 27).

[6] Periklis Akritidis, Manuel Costa, Miguel Castro and Steven Hand. 'Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense Against Out-of-Bounds Errors'. In: *Proceedings of the 18th Conference on USENIX Security Symposium* (Montreal, Canada). Berkeley, CA, USA: USENIX Association, 2009, pp. 51–66. URL: http://dl.acm.org/citation.cfm?id=1855768.1855772 (cit. on pp. 16, 29, 166).

[7] James P. Anderson. *Computer Security Technology Planning Study, Volume II*. ESD-TR-73-51. L. G. Hanscom Field, Bedford, MA 01730: Electronic Systems Division, Air Force Systems Command, Oct. 1972. URL: https://apps.dtic.mil/docs/citations/AD0772806 (cit. on p. 25).

[8] Android Open Source Project. *Control Flow Integrity*. URL: https://source.android.com/devices/tech/debug/cfi (visited on 16/08/2019) (cit. on p. 15).

[9] Android Open Source Project. *HWAddressSanitizer*. URL: https://source.android.com/devices/tech/debug/hwasan (visited on 20/09/2019) (cit. on p. 165).

[10] Jason Ansel, Petr Marchenko, Úlfar Erlingsson, Elijah Taylor, Brad Chen, Derek L. Schuff, David Sehr, Cliff L. Biffle and Bennet Yee. 'Language-Independent Sandboxing of Just-in-Time Compilation and Self-Modifying Code'. In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA). New York, NY, USA: ACM, 2011, pp. 355–366. DOI: 10.1145/1993498.1993540 (cit. on p. 29).

[11] ARM. *Procedure Call Standard for the ARM 64-Bit Architecture (AArch64)*. ARM IHI 0055B. 22nd May 2013. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ihi0055b/IHI0055B_aapcs64.pdf (cit. on p. 77).

[12] Arm Ltd. *ARM Cortex-A Series Programmer's Guide for ARMv8-A*. 2015. URL: https://static.docs.arm.com/den0024/a/DEN0024A_v8_architecture_PG.pdf (cit. on p. 70).

[13] Arm Ltd. *Arm Morello Program*. 2020. URL: https://developer.arm.com/architectures/cpu-architecture/a-profile/morello (cit. on pp. 31, 171).

[14] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami and Peter Sewell. 'ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS'. In: *Proceedings of the ACM on Programming Languages* (POPL Jan. 2019), 71:1–71:31. DOI: 10.1145/3290384 (cit. on pp. 99, 207).

[15] Simon Atanasyan. *D31528 [ELF][MIPS] Multi-GOT Implementation*. 31st Mar. 2017. URL: https://reviews.llvm.org/D31528 (visited on 19/08/2019) (cit. on p. 80).

[16] Todd M. Austin, Scott E. Breach, Gurindar S. Sohi, Todd M. Austin, Scott E. Breach and Gurindar S. Sohi. 'Efficient Detection of All Pointer and Array Access Errors'. In: *ACM SIGPLAN Notices* 29.6 (8th Jan. 1994), pp. 290–301. DOI: 10.1145/178243.178446 (cit. on p. 145).

[17] Leonid Azriel, Lukas Humbel, Reto Achermann, Alex Richardson, Moritz Hoffmann, Avi Mendelson, Timothy Roscoe, Robert N. M. Watson, Paolo Faraboschi and Dejan Milojicic. 'Memory-Side Protection With a Capability Enforcement Co-Processor'. In: *ACM Transactions on Architecture and Code Optimization* 16.1 (Mar. 2019), 5:1–5:26. DOI: 10.1145/3302257 (cit. on p. 21).

[18] Adam Barth, Charles Reis, Collin Jackson and Google Chrome Team. *The Security Architecture of the Chromium Browser*. 2008. URL: https://seclab.stanford.edu/websec/chromium/chromium-security-architecture.pdf (cit. on p. 170).

[19] Joe Bialek. 'The Evolution of CFI Attacks and Defenses'. OffensiveCon 2018. 17th Feb. 2018. URL: https://www.offensivecon.org/speakers/2018/joe-bialek.html (cit. on pp. 16, 27).

[20] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières and Dan Boneh. 'Hacking Blind'. In: *2014 IEEE Symposium on Security and Privacy*. 2014 IEEE Symposium on Security and Privacy. Washington, DC, USA: IEEE Computer Society, May 2014, pp. 227–242. DOI: 10.1109/SP.2014.22 (cit. on p. 26).

[21] Jasmin Blanchette and Mark Summerfield. 'A Brief History of Qt'. In: *C++ GUI Programming with Qt 4*. Prentice Hall, 21st June 2006. ISBN: 978-0-13-187249-3. URL: https://my.safaribooksonline.com/0131872494/pref04 (cit. on p. 153).

[22] Dion Blazakis. 'Interpreter Exploitation: Pointer Inference and JIT Spraying'. In: *BlackHat DC 2010*. Arlington, VA, USA, 31st Jan.–3rd Feb. 2010. URL: http://www.semantiscope.com/research/BHDC2010/BHDC-2010-Paper.pdf (cit. on p. 26).

[23] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh and Zhenkai Liang. 'Jump-Oriented Programming: A New Class of Code-Reuse Attack'. In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security - ASIACCS '11*. The 6th ACM Symposium. Hong Kong, China: ACM, 2011, p. 30. DOI: 10.1145/1966913.1966919 (cit. on p. 26).

[24] Tim Boland and Paul E. Black. 'Juliet 1.1 C/C++ and Java Test Suite'. In: *IEEE Computer* 45.10 (Oct. 2012), pp. 88–90. DOI: 10.1109/MC.2012.345 (cit. on p. 136).

[25] Erik Bosman and Herbert Bos. 'Framing Signals - A Return to Portable Shellcode'. In: *2014 IEEE Symposium on Security and Privacy* (San Jose, CA, USA). IEEE, May 2014, pp. 243–258. DOI: 10.1109/SP.2014.23 (cit. on p. 26).

[26] Erik Buchanan, Ryan Roemer, Hovav Shacham and Stefan Savage. 'When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC'. In: *Proceedings of the 15th ACM Conference on Computer and Communications Security* (Alexandria, Virginia, USA). New York, NY, USA: ACM, 2008, pp. 27–38. DOI: 10.1145/1455770.1455776 (cit. on p. 26).

[27] George Burgess. *FORTIFY in Android*. 13th Apr. 2017. URL: https://android-developers.googleblog.com/2017/04/fortify-in-android.html (visited on 25/07/2019) (cit. on pp. 137, 144).

[28] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler and Mathias Payer. 'Control-Flow Integrity: Precision, Security, and Performance'. In: *ACM Comput. Surv.* 50.1 (Apr. 2017), 16:1–16:33. DOI: 10.1145/3054924 (cit. on pp. 16, 109, 167).

[29] *C++ ABI for Itanium: Exception Handling*. 14th Mar. 2017. URL: https://itanium-cxx-abi.github.io/cxx-abi/abi-eh.html (visited on 22/07/2019) (cit. on p. 136).

[30] c0ntex. *How to Hijack the Global Offset Table with Pointers for Root Shells*. 9th Mar. 2006. URL: https://web.archive.org/web/20061112203558/http://www.milw0rm.com/papers/3 (visited on 10/09/2019) (cit. on p. 89).

[31] Canonical Ltd. *SecurityTeam/KnowledgeBase/SpectreAndMeltdown - Ubuntu Wiki*. URL: https://wiki.ubuntu.com/SecurityTeam/KnowledgeBase/SpectreAndMeltdown (visited on 26/08/2019) (cit. on p. 16).

[32] Nicholas P. Carter, Stephen W. Keckler and William J. Dally. 'Hardware Support for Fast Capability-Based Addressing'. In: *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA). New York, NY, USA: ACM, 1994, pp. 319–327. DOI: 10.1145/195473.195579 (cit. on p. 99).

[33] Miguel Castro, Manuel Costa and Tim Harris. 'Securing Software by Enforcing Data-Flow Integrity'. In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, Washington). Berkeley, CA, USA: USENIX Association, 2006, pp. 147–160. ISBN: 978-1-931971-47-8. URL: http://dl.acm.org/citation.cfm?id=1298455.1298470 (cit. on p. 28).

[34] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham and Richard Black. 'Fast Byte-Granularity Software Fault Isolation'. In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles - SOSP '09*. The ACM SIGOPS 22nd Symposium. Big Sky, Montana, USA: ACM, 2009, p. 45. DOI: 10.1145/1629575.1629581 (cit. on pp. 27, 29).

[35] Satish Chandra and Thomas Reps. 'Physical Type Checking for C'. In: *Proceedings of the 1999 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Toulouse, France). New York, NY, USA: ACM, 1999, pp. 66–75. DOI: 10.1145/316158.316183 (cit. on p. 129).

[36] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham and Marcel Winandy. 'Return-Oriented Programming without Returns'. In: *Proceedings of the 17th ACM Conference on Computer and Communications Security - CCS '10*. The 17th ACM Conference. Chicago, Illinois, USA: ACM, 2010, p. 559. DOI: 10.1145/1866307.1866370 (cit. on p. 26).

[37] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar and Ravishankar K. Iyer. 'Non-Control-Data Attacks Are Realistic Threats'. In: *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14* (Baltimore, MD). Berkeley, CA, USA: USENIX Association, 2005, pp. 12–12. URL: http://dl.acm.org/citation.cfm?id=1251398.1251410 (cit. on p. 28).

[38] Xi Chen, Asia Slowinska and Herbert Bos. 'On the Detection of Custom Memory Allocators in C Binaries'. In: *Empirical Software Engineering* 21.3 (1st June 2016), pp. 753–777. DOI: 10.1007/s10664-015-9362-z (cit. on p. 54).

[39] Long Cheng, Hans Liljestrand, Thomas Nyman, Yu Tsung Lee, Danfeng Yao, Trent Jaeger and N. Asokan. 'Exploitation Techniques and Defenses for Data-Oriented Attacks'. In: (21st Feb. 2019). arXiv: 1902.08359 [cs]. URL: http://arxiv.org/abs/1902.08359 (cit. on p. 28).

[40] David Chisnall, Brooks Davis, Khilan Gudka, David Brazdil, Alexandre Joannou, Jonathan Woodruff, A. Theodore Markettos, J. Edward Maste, Robert Norton, Stacey Son, Michael Roe, Simon W. Moore, Peter G. Neumann, Ben Laurie and Robert N. M. Watson. 'CHERI JNI: Sinking the Java Security Model into the C'. In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China). New York, NY, USA: ACM, 2017, pp. 569–583. DOI: 10.1145/3037697.3037725 (cit. on pp. 17, 31).

[41] David Chisnall, Colin Rothwell, Robert N. M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis and Peter G. Neumann. 'Beyond the PDP-11: Architectural Support for a Memory-Safe C Abstract Machine'. In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. Istanbul, Turkey: ACM, Mar. 2015, pp. 117–130. DOI: 10.1145/2694344.2694367 (cit. on pp. 17, 38, 48, 52, 56, 58, 126, 150, 157).

[42] Jessica Clarke. 'CHERI: Minimising in-address-space privilege'. BA Thesis. Cambridge, UK: University of Cambridge, 19th May 2017 (cit. on pp. 81, 82, 100, 101).

[43] Kees Cook. *-fstack-protector-strong*. 27th Jan. 2014. URL: https://outflux.net/blog/archives/2014/01/27/fstack-protector-strong/ (visited on 02/08/2019) (cit. on pp. 25, 133).

[44] Jonathan Corbet. *Vmsplice(): The Making of a Local Root Exploit*. 12th Feb. 2008. URL: https://lwn.net/Articles/268783/ (visited on 30/09/2019) (cit. on p. 111).

[45] MITRE Corporation. *2019 CWE Top 25 Most Dangerous Software Errors*. 18th Sept. 2019. URL: https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html (visited on 20/09/2019) (cit. on pp. 15, 25, 39, 169).

[46] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle and Qian Zhang. 'StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks'. In: *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*. 7th USENIX Security Symposium (San Antonio, Texas). San Antonio, Texas: USENIX Association, 26th–29th Jan. 1998. URL: https://www.usenix.org/legacy/publications/library/proceedings/sec98/full_papers/cowan/cowan.pdf (cit. on pp. 16, 25, 76, 137).

[47] *CTSRD-CHERI/Bodiagsuite: A Rescued Copy of the Old Bodiagsuite Corpus and Some Minimal Tooling*. Capability Hardware Enhanced RISC Instructions, 26th July 2019. URL: https://github.com/CTSRD-CHERI/bodiagsuite (visited on 27/07/2019) (cit. on p. 137).

[48] *CTSRD-CHERI/Postgres*. URL: https://github.com/CTSRD-CHERI/postgres (visited on 04/09/2019) (cit. on p. 167).

[49] John Dallman. *[Cfe-Users] Stack Buffer Overflow Protection*. E-mail. 30th Sept. 2015. URL: http://lists.llvm.org/pipermail/cfe-users/2015-September/000796.html (visited on 16/08/2019) (cit. on pp. 16, 26).

[50] Thurston H. Y. Dang, Petros Maniatis and David Wagner. 'The Performance Cost of Shadow Stacks and Stack Canaries'. In: *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security - ASIA CCS '15*. The 10th ACM Symposium. Singapore, Republic of Singapore: ACM, 2015, pp. 555–566. DOI: 10.1145/2714576.2714635 (cit. on pp. 16, 25, 167).

[51] Al Danial. *CLOC*. 3rd Sept. 2019. URL: https://github.com/AlDanial/cloc (visited on 03/09/2019) (cit. on pp. 147, 149).

[52] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann and Fabian Monrose. 'Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection'. In: *Proceedings of the 23rd USENIX Conference on Security Symposium* (San Diego, CA). Berkeley, CA, USA: USENIX Association, 2014, pp. 401–416. ISBN: 978-1-931971-15-7. URL: http://dl.acm.org/citation.cfm?id=2671225.2671251 (cit. on p. 27).

[53] Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son and Jonathan Woodruff. *CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-Time Environment*. Technical Report UCAM-CL-TR-932. University of Cambridge, Computer Laboratory, 2019. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-932.html (cit. on pp. 20, 38, 39, 42, 44, 50, 53, 70, 74, 111, 151, 156, 164).

[54] Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son and Jonathan Woodruff. 'CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment'. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA). New York, NY, USA: ACM, 2019, pp. 379–393. DOI: 10.1145/3297858.3304042 (cit. on pp. 17, 19, 20, 31, 38, 39, 42, 74, 100, 111, 113, 136, 137, 140, 148, 150, 163, 164).

[55] Department for Business, Energy & Industrial Strategy, UK Research and Innovation and Greg Clark. *Global Businesses – Including Google and Microsoft – Back UK to Block Cyber Threats with New Tech*. 22nd July 2019. URL: https://www.gov.uk/government/news/global-businesses-including-google-and-microsoft-back-uk-to-block-cyber-threats-with-new-tech (visited on 25/07/2019) (cit. on pp. 18, 171).

[56] Department of Heath & Social Care. *Securing Cyber Resilience in Health and Care: Progress Update October 2018*. Oct. 2018. URL: https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/747464/securing-cyber-resilience-in-health-and-care-september-2018-update.pdf (cit. on p. 15).

[57] Solar Designer. *Linux Kernel Patch from the Openwall Project*. URL: https://www.openwall.com/linux/README.shtml (visited on 30/09/2019) (cit. on p. 26).

[58] Joe Devietti, Colin Blundell, Milo M. K. Martin and Steve Zdancewic. 'Hardbound: Architectural Support for Spatial Safety of the C Programming Language'. In: *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (Seattle, WA, USA). New York, NY, USA: ACM, 2008, pp. 103–114. DOI: 10.1145/1346281.1346295 (cit. on pp. 29, 136, 143, 160, 165).

[59] Dinakar Dhurjati and Vikram Adve. 'Backwards-Compatible Array Bounds Checking for C with Very Low Overhead'. In: *Proceedings of the 28th International Conference on Software Engineering* (Shanghai, China). New York, NY, USA: ACM, 2006, pp. 162–171. DOI: 10.1145/1134285.1134309 (cit. on p. 160).

[60] Dinakar Dhurjati, Sumant Kowshik and Vikram Adve. 'SAFECode: Enforcing Alias Analysis for Weakly Typed Languages'. In: *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada). New York, NY, USA: ACM, 2006, pp. 144–157. DOI: 10.1145/1133981.1133999 (cit. on p. 160).

[61] Alessandro Di Federico, Amat Cama, Yan Shoshitaishvili, Christopher Kruegel and Giovanni Vigna. 'How the ELF Ruined Christmas'. In: *Proceedings of the 24th USENIX Conference on Security Symposium* (Washington, D.C.). Berkeley, CA, USA: USENIX Association, 2015, pp. 643–658. ISBN: 978-1-931971-23-2. URL: http://dl.acm.org/citation.cfm?id=2831143.2831184 (cit. on p. 75).

[62] Ulrich Drepper. *ELF Handling For Thread-Local Storage*. Version 0.21. 22nd Aug. 2013. URL: https://www.akkadia.org/drepper/tls.pdf (cit. on p. 105).

[63] Gregory J. Duck and Roland H. C. Yap. 'EffectiveSan: Type and Memory Error Detection Using Dynamically Typed C/C++'. In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA). New York, NY, USA: ACM, 2018, pp. 181–195. DOI: 10.1145/3192366.3192388 (cit. on pp. 29, 137, 142).

[64] Gregory J. Duck and Roland H. C. Yap. 'Heap Bounds Protection with Low Fat Pointers'. In: *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain). New York, NY, USA: ACM, 2016, pp. 132–142. DOI: 10.1145/2892208.2892212 (cit. on pp. 29, 112, 138, 142, 165).

[65] Gregory J. Duck, Roland H. C. Yap and Lorenzo Cavallaro. 'Stack Bounds Protection with Low Fat Pointers'. In: *Proceedings 2017 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium. San Diego, CA: Internet Society, 2017. DOI: 10.14722/ndss.2017.23287 (cit. on pp. 16, 29, 112, 138, 142, 165).

[66] M. W. Eichin and J. A. Rochlis. 'With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988'. In: *Proceedings. 1989 IEEE Symposium on Security and Privacy*. Proceedings. 1989 IEEE Symposium on Security and Privacy. May 1989, pp. 326–343. DOI: 10.1109/SECPRI.1989.36307 (cit. on p. 25).

[67] Archibald Samuel Elliott, Andrew Ruef, Michael Hicks and David Tarditi. 'Checked C: Making C Safe by Extension'. In: *2018 IEEE Cybersecurity Development (SecDev)*. 2018 IEEE Cybersecurity Development (SecDev). Sept. 2018, pp. 53–60. DOI: 10.1109/SecDev.2018.00015 (cit. on pp. 16, 25, 145, 157, 165).

[68] Lawrence Esswood. 'CheriOS: A high-performance and completely untrusted single-address-space capability operating system'. PhD Thesis. Cambridge, UK: University of Cambridge, 2020. to be submitted (cit. on pp. 87, 105, 106, 171).

[69] Chris Evans. *Announcing Project Zero*. 15th July 2014. URL: https://security.googleblog.com/2014/07/announcing-project-zero.html (visited on 25/07/2019) (cit. on p. 15).

[70] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard and Hamed Okhravi. 'Missing the Point(Er): On the Effectiveness of Code Pointer Integrity'. In: *2015 IEEE Symposium on Security and Privacy*. 2015 IEEE Symposium on Security and Privacy (SP). San Jose, CA: IEEE, May 2015, pp. 781–796. DOI: 10.1109/SP.2015.53 (cit. on p. 28).

[71] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi and Stelios Sidiroglou-Douskos. 'Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity'. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*. The 22nd ACM SIGSAC Conference. Denver, Colorado, USA: ACM, 2015, pp. 901–913. DOI: 10.1145/2810103.2813646 (cit. on p. 27).

[72] Jason Evans. 'A Scalable Concurrent malloc(3) Implementation for FreeBSD'. In: BSDCan. 16th Apr. 2006. URL: https://www.bsdcan.org/2006/papers/jemalloc.pdf (cit. on pp. 70, 135, 160).

[73] R. S. Fabry. 'Capability-Based Addressing'. In: *Commun. ACM* 17.7 (July 1974), pp. 403–412. DOI: 10.1145/361011.361070 (cit. on p. 30).

[74] Nathaniel Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, Theo Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann and Robert Watson. 'Cornucopia: Temporal Safety for CHERI Heaps'. In: *2020 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2020, pp. 1507–1524. DOI: 10.1109/SP40000.2020.00098 (cit. on pp. 17, 21, 30–32, 38, 39, 170, 171).

[75] Philip J. Fleming and John J. Wallace. 'How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results'. In: *Communications of the ACM* 29.3 (1st Mar. 1986), pp. 218–221. DOI: 10.1145/5666.5673 (cit. on p. 166).

[76] Free Software Foundation. *GCC 9 Release Series — Changes, New Features, and Fixes*. URL: https://gcc.gnu.org/gcc-9/changes.html (visited on 16/07/2019) (cit. on pp. 16, 145).

[77] Free Software Foundation. *Using the GNU Compiler Collection (GCC): Arrays of Length Zero*. URL: https://gcc.gnu.org/onlinedocs/gcc/Zero-Length.html (visited on 30/07/2019) (cit. on p. 120).

[78] Free Software Foundation. *Using the GNU Compiler Collection (GCC): Arrays of Variable Length*. URL: https://gcc.gnu.org/onlinedocs/gcc/Variable-Length.html (visited on 29/07/2019) (cit. on p. 120).

[79] Free Software Foundation. *Using the GNU Compiler Collection (GCC): Common Type Attributes*. URL: https://gcc.gnu.org/onlinedocs/gcc-8.3.0/gcc/Common-Type-Attributes.html (visited on 31/07/2019) (cit. on pp. 119, 221).

[80] Free Software Foundation. *Using the GNU Compiler Collection (GCC): Object Size Checking Built-in Functions*. URL: https://gcc.gnu.org/onlinedocs/gcc/Object-Size-Checking.html (visited on 19/07/2019) (cit. on p. 143).

[81] Ronald Gil. 'The Undefined Quest for Full Memory Safety'. MEng Thesis. Cambridge, MA, USA: Massachusetts Institute of Technology, Feb. 2018. URL: https://dspace.mit.edu/handle/1721.1/119551 (cit. on p. 141).

[82] Ronald Gil, Hamed Okhravi and Howard E. Shrobe. 'There's a Hole in the Bottom of the C: On the Effectiveness of Allocation Protection'. In: *2018 IEEE Cybersecurity Development (SecDev)*. 2018 IEEE Cybersecurity Development (SecDev). Sept. 2018, pp. 102–109. DOI: 10.1109/SecDev.2018.00021 (cit. on pp. 112, 133, 141).

[83] Olivier Goffart. *QStringLiteral Explained*. 21st May 2012. URL: https://woboq.com/blog/qstringliteral.html (visited on 04/07/2019) (cit. on pp. 69, 153).

[84] Enes Göktaş, Angelos Economopoulos, Robert Gawlik, Elias Athanasopoulos, Georgios Portokalidis and Herbert Bos. 'Bypassing Clang's SafeStack for Fun and Profit'. Black Hat Europe 2016 (London, United Kindom). 1st–4th Nov. 2016. URL: https://www.blackhat.com/eu-16/briefings.html#bypassing-clangs-safestack-for-fun-and-profit (cit. on p. 28).

[85] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos and Cristiano Giuffrida. 'ASLR on the Line: Practical Cache Attacks on the MMU'. In: *Proceedings 2017 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium (San Diego, CA). Internet Society, 2017. DOI: 10.14722/ndss.2017.23271 (cit. on p. 27).

[86] Matthew Gretton-Dann. *Arm Architecture Armv8.5-A Announcement*. 17th Sept. 2018. URL: https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/arm-a-profile-architecture-2018-developments-armv85a (visited on 08/08/2019) (cit. on pp. 16, 27, 106).

[87] Richard Grisenthwaite. *A Safer Digital Future, By Design*. Library Catalog: www.arm.com. 17th Oct. 2019. URL: https://www.arm.com/blogs/blueprint/digital-security-by-design (visited on 07/04/2020) (cit. on p. 171).

[88] Richard Grisenthwaite. *Arm Supports UK in Becoming a Leading Global Cybersecurity Player*. 28th Jan. 2019. URL: https://community.arm.com/blog/company/b/blog/posts/supporting-the-uk-in-becoming-a-leading-global-player-in-cybersecurity (visited on 25/07/2019) (cit. on p. 18).

[89] Samuel Groß. 'Attacking JavaScript Engines: A Case Study of JavaScriptCore and CVE-2016-4622'. In: *Phrack Magazine* (Paper Feed 27th Oct. 2016). URL: http://www.phrack.org/papers/attacking_javascript_engines.html (cit. on p. 154).

[90] Nadav Grossman. *EternalBlue - Everything There Is To Know*. 29th Sept. 2017. URL: https://research.checkpoint.com/eternalblue-everything-know/ (visited on 08/07/2019) (cit. on p. 15).

[91] Daniel Gruss, Dave Hansen and Brendan Gregg. 'Kernel Isolation: From an Academic Idea to an Efficient Patch for Every Computer'. In: *;login: the USENIX Magazine* 43.4 (Dec. 2018), pp. 10–14. URL: https://www.usenix.org/system/files/login/articles/login_winter18_03_gruss.pdf (cit. on pp. 16, 29, 167).

[92] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice and Stefan Mangard. 'KASLR Is Dead: Long Live KASLR'. In: *Engineering Secure Software and Systems*. Ed. by Eric Bodden, Mathias Payer and Elias Athanasopoulos. Springer International Publishing, 2017, pp. 161–176. DOI: 10.1007/978-3-319-62105-0_11 (cit. on pp. 16, 29).

[93] David Gudeman. *Representing Type Information in Dynamically Typed Languages*. TR 93-27. The University of Arizona, Oct. 1993. URL: https://www.cs.arizona.edu/sites/default/files/TR93-27.pdf (cit. on pp. 41, 66, 148, 154).

[94] Khilan Gudka, Alexander Richardson and Robert N. M. Watson. 'Protecting C++ Applications Using CHERI'. Principles of Secure Compilation (PriSC) 2019 (Cascais, Portugal). 13th Jan. 2019. URL: https://popl19.sigplan.org/details/prisc-2019/6/Protecting-C-Applications-Using-CHERI (cit. on pp. 20, 154).

[95] Khilan Gudka, Robert N. M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann and Alex Richardson. 'Clean Application Compartmentalization with SOAAP'. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA). New York, NY, USA: ACM, 12th–16th Oct. 2015, pp. 1016–1031. DOI: 10.1145/2810103.2813611 (cit. on pp. 18, 21).

[96] Khilan Gudka, Robert N. M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann and Alex Richardson. *Clean Application Compartmentalization with SOAAP (Extended Version)*. Technical Report UCAM-CL-TR-873. University of Cambridge, Computer Laboratory, Aug. 2015. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-873.html (cit. on pp. 18, 21).

[97] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge and R. B. Brown. 'MiBench: A Free, Commercially Representative Embedded Benchmark Suite'. In: *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*. Proceedings of the Fourth

Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538). Dec. 2001, pp. 3–14. DOI: 10.1109/WWC.2001.990739 (cit. on p. 160).

[98] Troy D. Hanson. *Uthash: A Hash Table for C Structures*. URL: https://troydhanson.github.io/uthash/ (visited on 21/07/2019) (cit. on p. 125).

[99] Niranjan Hasabnis, Ashish Misra and R. Sekar. 'Light-Weight Bounds Checking'. In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization* (San Jose, California). New York, NY, USA: ACM, 2012, pp. 135–144. DOI: 10.1145/2259016.2259034 (cit. on pp. 29, 166).

[100] John L. Henning. 'SPEC CPU2006 Benchmark Descriptions'. In: *SIGARCH Comput. Archit. News* 34.4 (Sept. 2006), pp. 1–17. DOI: 10.1145/1186736.1186737 (cit. on p. 161).

[101] John L. Henning. 'SPEC CPU2006 Memory Footprint'. In: *SIGARCH Comput. Archit. News* 35.1 (Mar. 2007), pp. 84–89. DOI: 10.1145/1241601.1241618 (cit. on p. 161).

[102] Hewlett-Packard. *64-Bit Runtime Architecture for PA-RISC 2.0*. Version 3.3. 6th Oct. 1997. URL: https://parisc.wiki.kernel.org/images-parisc/5/59/Pa64rt.pdf (cit. on pp. 81, 90).

[103] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena and Zhenkai Liang. 'Automatic Generation of Data-Oriented Exploits'. In: *Proceedings of the 24th USENIX Conference on Security Symposium* (Washington, D.C.). Berkeley, CA, USA: USENIX Association, 2015, pp. 177–192. ISBN: 978-1-931971-23-2. URL: http://dl.acm.org/citation.cfm?id=2831143.2831155 (cit. on p. 28).

[104] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena and Zhenkai Liang. 'Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks'. In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016 IEEE Symposium on Security and Privacy (SP). May 2016, pp. 969–986. DOI: 10.1109/SP.2016.62 (cit. on p. 28).

[105] Larry Huffman and David Graves. *MIPSpro™ Assembly Language Programmer's Guide*. manual 007–2418–006. Silicon Graphics, Inc., June 2003. URL: https://irix7.com/techpubs/007-2418-006.pdf (cit. on p. 79).

[106] IBM. *Function Pointer Comparison in DLL Code*. URL: https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.4.0/com.ibm.zos.v2r4.cbcpx01/cbc1p2224.htm (visited on 26/07/2019) (cit. on p. 91).

[107] UK Reasearch and Innovation. *Securing the Future of the Digital Economy*. July 2019. URL: https://www.ukri.org/innovation/industrial-strategy-challenge-fund/digital-security-by-design/securing-the-future-of-the-digital-economy/ (visited on 25/07/2019) (cit. on pp. 15, 171).

[108] Intel. *Control-Flow Enforcement Technology Specification*. 334525-003, Revision 3.0. May 2019. URL: https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf (cit. on pp. 16, 27, 76, 106).

[109] Intel. *Intel Itanium Processor-Specific Application Binary Interface (ABI)*. 245370-003. May 2001. URL: http://refspecs.linuxfoundation.org/elf/IA64-SysV-psABI.pdf (cit. on pp. 81, 90).

[110] Intel. *Introduction to Intel® Memory Protection Extensions*. 16th July 2013. URL: https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions (visited on 25/06/2019) (cit. on pp. 16, 29).

[111] International standardization working group for the programming language C++. *Working Draft, Standard for Programming Language C++*. ISO/IEC, 21st Mar. 2017. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf (cit. on pp. 44, 116).

[112] International standardization working group for the programming language C. *Programming Language - C (Committee Draft)*. ISO/IEC 9899:201x. ISO/IEC, 12th Apr. 2011. URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf (cit. on pp. 38, 47, 52, 69, 70, 90, 91, 93).

[113] International standardization working group for the programming language C. *Programming Language - C (Committee Draft)*. ISO/IEC 9899:202x. ISO/IEC, 13th Mar. 2019. URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2346.pdf (cit. on pp. 41, 42, 44, 118, 120, 122, 129).

[114] Kyriakos Ispoglou, Bader AlBassam, Trent Jaeger and Mathias Payer. 'Block Oriented Programming: Automating Data-Only Attacks'. In: (12th May 2018). arXiv: 1805.04767 [cs]. URL: http://arxiv.org/abs/1805.04767 (cit. on p. 28).

[115] Jakub Jelinek. *[PATCH] Object Size Checking to Prevent (Some) Buffer Overflows*. 21st Sept. 2004. URL: https://gcc.gnu.org/ml/gcc-patches/2004-09/msg02055.html (visited on 19/07/2019) (cit. on pp. 137, 144).

[116] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney and Yanling Wang. 'Cyclone: A Safe Dialect of C'. In: *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2002, pp. 275–288. ISBN: 978-1-880446-00-3. URL: http://dl.acm.org/citation.cfm?id=647057.713871 (cit. on pp. 16, 25, 157, 165).

[117] Alexandre Joannou, Jonathan Woodruff, Robert Kovacsics, Simon W. Moore, Alex Bradbury, Hongyan Xia, Robert N. M. Watson, David Chisnall, Michael Roe, Brooks Davis, Edward Napierala, John Baldwin, Khilan Gudka, Peter G. Neumann, Alfredo Mazzinghi, Alex Richardson, Stacey Son and A Theodore Markettos. 'Efficient Tagged Memory'. In: *2017 IEEE International Conference on Computer Design*

(ICCD). 2017 IEEE International Conference on Computer Design (ICCD). 1st Nov. 2017, pp. 641–648. DOI: 10.1109/ICCD.2017.112 (cit. on pp. 17, 21, 30, 31, 38).

[118] Alexandre J. P. Joannou. *High-Performance Memory Safety: Optimizing the CHERI Capability Machine*. Technical Report UCAM-CL-TR-936. University of Cambridge, Computer Laboratory, 2019. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-936.html (cit. on pp. 51, 55).

[119] Stephen Kell. 'Dynamically Diagnosing Type Errors in Unsafe Code'. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands). New York, NY, USA: ACM, 2016, pp. 800–819. DOI: 10.1145/2983990.2983998 (cit. on p. 29).

[120] Stephen Kell, Dominic P. Mulligan and Peter Sewell. 'The Missing Link: Explaining ELF Static Linking, Semantically'. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands). New York, NY, USA: ACM, 2016, pp. 607–623. DOI: 10.1145/2983990.2983996 (cit. on p. 33).

[121] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz and Yuval Yarom. 'Spectre Attacks: Exploiting Speculative Execution'. In: (3rd Jan. 2018). arXiv: 1801.01203 [cs]. URL: http://arxiv.org/abs/1801.01203 (cit. on pp. 16, 28, 108).

[122] Andrew Koenig. *C Traps and Pitfalls*. 01 edition. Reading, Mass: Addison Wesley, 1989. 160 pp. ISBN: 978-0-201-17928-6 (cit. on p. 153).

[123] Maria Korolov. *WannaCry Fallout – the Worst Is yet to Come, Experts Say*. 17th May 2017. URL: https://www.csoonline.com/article/3196400/wannacry-fallout-the-worst-is-yet-to-come-experts-say.html (visited on 25/08/2019) (cit. on p. 15).

[124] Kendra Kratkiewicz and Richard Lippmann. 'Using a Diagnostic Corpus of C Programs to Evaluate Buffer Overflow Detection by Static Analysis Tools'. In: *Bugs: Workshop on the Evaluation of Software Defect Detection Tools*. 12th June 2005. URL: http://www.cs.umd.edu/~pugh/BugWorkshop05/papers/62-kratkiewicz.pdf (cit. on p. 137).

[125] Kendra June Kratkiewicz. 'Evaluating Static Analysis Tools for Detecting Buffer Overflows in C Code'. Master of Liberal Arts Thesis. Cambridge, MA, USA: Harvard University, Mar. 2005 (cit. on p. 136).

[126] Greg Kroah-Hartman. *Container_of()*. 18th Feb. 2005. URL: http://www.kroah.com/log/linux/container_of.html (visited on 21/07/2019) (cit. on p. 125).

[127] Greg Kroah-Hartman. 'Driving Me Nuts: The Driver Model Core, Part I'. In: *Linux J.* 2003.110 (June 2003), pp. 7–. URL: http://dl.acm.org/citation.cfm?id=774727.774734 (cit. on p. 125).

[128] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos and Cristiano Giuffrida. 'Delta Pointers: Buffer Overflow Checks without the Checks'. In: *Proceedings of the Thirteenth EuroSys Conference on - EuroSys '18*. The Thirteenth EuroSys Conference. Porto, Portugal: ACM, 2018, pp. 1–14. DOI: 10.1145/3190508.3190553 (cit. on pp. 29, 165).

[129] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber and Christof Fetzer. 'SGXBOUNDS: Memory Safety for Shielded Execution'. In: *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia). New York, NY, USA: ACM, Apr. 2017, pp. 205–221. DOI: 10.1145/3064176.3064192 (cit. on pp. 29, 165).

[130] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea and Dawn Song. 'Poster: Getting The Point(Er): On the Feasibility of Attacks on Code-Pointer Integrity'. In: *IEEE Symposium on Security and Privacy* (San Jose, CA). May 2015 (cit. on p. 28).

[131] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar and Dawn Song. 'Code-Pointer Integrity'. In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO). Berkeley, CA, USA: USENIX Association, 2014, pp. 147–163. ISBN: 978-1-931971-16-4. URL: http://dl.acm.org/citation.cfm?id=2685048.2685061 (cit. on pp. 27, 28).

[132] Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight Jr. and Andre DeHon. 'Low-Fat Pointers: Compact Encoding and Efficient Gate-Level Implementation of Fat Pointers for Spatial Safety and Capability-Based Security'. In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (Berlin, Germany). New York, NY, USA: ACM, 2013, pp. 721–732. DOI: 10.1145/2508859.2516713 (cit. on pp. 29, 52, 165).

[133] Bingchen Lan, Yan Li, Hao Sun, Chao Su, Yao Liu and Qingkai Zeng. 'Loop-Oriented Programming: A New Code Reuse Attack to Bypass Modern Defenses'. In: *Proceedings of the 2015 IEEE Trustcom/BigDataSE/ISPA*. Washington, DC, USA: IEEE Computer Society, 20th–22nd Aug. 2015, pp. 190–197. DOI: 10.1109/Trustcom-BigDataSe-ISPA.2015.374 (cit. on p. 26).

[134] Chris Lattner and Vikram Adve. 'LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation'. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime*

Optimization (Palo Alto, California). Washington, DC, USA: IEEE Computer Society, Mar. 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665 (cit. on p. 16).

[135] Nika Layzell. *Fission Engineering Newsletter #1*. 4th Feb. 2019. URL: https://mystor.github.io/fission-news-1.html (visited on 17/09/2019) (cit. on p. 170).

[136] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado and Brent Byunghoon Kang. 'Hacking in Darkness: Return-Oriented Programming Against Secure Enclaves'. In: *Proceedings of the 26th USENIX Conference on Security Symposium* (Vancouver, BC, Canada). Berkeley, CA, USA: USENIX Association, 2017, pp. 523–539. ISBN: 978-1-931971-40-9. URL: http://dl.acm.org/citation.cfm?id=3241189.3241231 (cit. on p. 26).

[137] John R. Levine. *Linkers and Loaders*. San Francisco: Morgan Kaufmann Publishers Inc., 10th Jan. 1999. 256 pp. ISBN: 978-1-55860-496-4. URL: http://dl.acm.org/citation.cfm?id=519563 (cit. on pp. 33–35, 76, 88).

[138] Henry M. Levy. *Capability-Based Computer Systems*. Newton, MA, USA: Butterworth-Heinemann, 1984. ISBN: 978-0-932376-22-0. URL: https://homes.cs.washington.edu/~levy/capabook/ (cit. on p. 30).

[139] Paul Liétar, Theodore Butler, Sylvan Clebsch, Sophia Drossopoulou, Juliana Franco, Matthew J. Parkinson, Alex Shamis, Christoph M. Wintersteiger and David Chisnall. 'Snmalloc: A Message Passing Allocator'. In: *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management* (Phoenix, AZ, USA). New York, NY, USA: ACM, 2019, pp. 122–135. DOI: 10.1145/3315573.3329980 (cit. on p. 42).

[140] Kewen Lin and Jinsong Ji. *Improve the Performance of Function Calls with OpenPOWER ABI*. 10th July 2015, p. 9. URL: https://www.ibm.com/developerworks/library/l-improve-performance-openpower-abi/l-improve-performance-openpower-abi-pdf.pdf (cit. on pp. 107, 108).

[141] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom and Mike Hamburg. 'Meltdown'. In: (3rd Jan. 2018). arXiv: 1801.01207 [cs]. URL: http://arxiv.org/abs/1801.01207 (cit. on pp. 16, 28, 108).

[142] LLVM Foundation. *LLVM "test-suite" Repository*. 5th June 2019. URL: https://github.com/llvm/llvm-test-suite (visited on 25/06/2019) (cit. on p. 167).

[143] Robert Lougher. *43308 – StackProtector - Stack Violation Not Caught*. 13th Sept. 2019. URL: https://bugs.llvm.org/show_bug.cgi?id=43308 (visited on 15/10/2019) (cit. on p. 133).

[144] Ivan Lozano. *Compiler-Based Security Mitigations in Android P*. 27th June 2018. URL: https://android-developers.googleblog.com/2018/06/compiler-based-security-mitigations-in.html (visited on 30/09/2019) (cit. on p. 27).

[145] H. J. Lu, Michael Matz, Milind Girkar, Jan Hubička, Andreas Jaeger and Mark Mitchell. *System V Application Binary Interface AMD64 Architecture Processor Supplement*. 28th Jan. 2018. URL: https://github.com/hjl-tools/x86-psABI/wiki/x86-64-psABI-1.0.pdf (cit. on p. 77).

[146] Stephen McCamant and Greg Morrisett. 'Evaluating SFI for a CISC Architecture'. In: *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15* (Vancouver, B.C., Canada). Berkeley, CA, USA: USENIX Association, 2006. URL: http://dl.acm.org/citation.cfm?id=1267336.1267351 (cit. on p. 29).

[147] Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson and Peter Sewell. 'Exploring C Semantics and Pointer Provenance'. In: *Proceedings of the ACM on Programming Languages* (POPL Jan. 2019), 67:1–67:32. DOI: 10.1145/3290380 (cit. on pp. 20, 38, 39, 69, 70).

[148] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson and Peter Sewell. 'Into the Depths of C: Elaborating the De Facto Standards'. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA). New York, NY, USA: ACM, 2016, pp. 1–15. DOI: 10.1145/2908080.2908081 (cit. on pp. 39, 47, 59, 72).

[149] Microsoft. */GS (Buffer Security Check)*. 4th Nov. 2016. URL: https://docs.microsoft.com/en-us/cpp/build/reference/gs-buffer-security-check (visited on 05/07/2019) (cit. on pp. 16, 26).

[150] Microsoft. *A Detailed Description of the Data Execution Prevention (DEP) Feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003*. 11th July 2017. URL: https://support.microsoft.com/en-gb/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in (visited on 30/09/2019) (cit. on p. 26).

[151] Microsoft. *Mitigating Spectre Variant 2 with Retpoline on Windows*. URL: https://techcommunity.microsoft.com/t5/Windows-Kernel-Internals/Mitigating-Spectre-variant-2-with-Retpoline-on-Windows/ba-p/295618 (visited on 14/08/2019) (cit. on pp. 16, 29).

[152] Microsoft. *Windows Kernel Macros - Windows Drivers*. 17th Oct. 2018. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/mm-bad-pointer#containing_record (visited on 11/07/2019) (cit. on p. 125).

[153] Matt Miller. 'Trends, Challenges, and Strategic Shifts in the Software Vulnerability Mitigation Landscape'. BlueHat IL 2019 (Tel Aviv). 7th Feb. 2019. URL: https://msrnd-cdn-stor.azureedge.net/bluehat/bluehatil/2019/assets/doc/Trends%2C%20Challenges%2C%20and%20Strategic%20Shifts%20in%20the%20Software%20Vulnerability%20Mitigation%20Landscape.pdf (cit. on pp. 15, 25, 26, 32, 169).

[154] Ingo Molnar. *[GIT PULL] X86/Cpu Changes for v5.4*. 16th Sept. 2019. URL: https://www.lkml.org/lkml/2019/9/16/331 (visited on 13/10/2019) (cit. on pp. 16, 145).

[155] Santosh Nagarakatte. *Experimental SoftBoundCETS for LLVM-3.9*. 18th Mar. 2019. URL: https://github.com/santoshn/SoftBoundCETS-3.9 (visited on 27/07/2019) (cit. on pp. 139, 143).

[156] Santosh Nagarakatte, Milo M. K. Martin and Steve Zdancewic. 'Everything You Want to Know About Pointer-Based Checking'. In: *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Ed. by Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner and Greg Morrisett. Vol. 32. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2015, pp. 190–208. DOI: 10.4230/LIPIcs.SNAPL.2015.190 (cit. on pp. 137, 143).

[157] Santosh Nagarakatte, Milo M. K. Martin and Steve Zdancewic. 'Watchdog: Hardware for Safe and Secure Manual Memory Management and Full Memory Safety'. In: *Proceedings of the 39th Annual International Symposium on Computer Architecture* (Portland, Oregon). Washington, DC, USA: IEEE Computer Society, June 2012, pp. 189–200. DOI: 10.1109/ISCA.2012.6237017 (cit. on pp. 29, 143, 165).

[158] Santosh Nagarakatte, Milo M. K. Martin and Steve Zdancewic. 'WatchdogLite: Hardware-Accelerated Compiler-Based Pointer Checking'. In: *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Orlando, FL, USA). New York, NY, USA: ACM, 2014, 175:175–175:184. DOI: 10.1145/2581122.2544147 (cit. on pp. 29, 143, 165).

[159] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin and Steve Zdancewic. 'CETS: Compiler Enforced Temporal Safety for C'. In: *Proceedings of the 2010 International Symposium on Memory Management* (Toronto, Ontario, Canada). New York, NY, USA: ACM, 2010, pp. 31–40. DOI: 10.1145/1806651.1806657 (cit. on p. 143).

[160] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin and Steve Zdancewic. 'SoftBound: Highly Compatible and Complete Spatial Memory Safety for C'. In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland). New York, NY, USA: ACM, 2009, pp. 245–258. DOI: 10.1145/1542476.1542504 (cit. on pp. 29, 137, 139, 143, 160, 165, 166).

[161] Santosh Ganapati Nagarakatte. 'Practical Low-Overhead Enforcement of Memory Safety for C Programs'. PhD Thesis. Philadelphia, PA, USA: University of Pennsylvania, 2012 (cit. on pp. 137, 143).

[162] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak and Westley Weimer. 'CCured: Type-Safe Retrofitting of Legacy Software'. In: *ACM Transactions on Programming Languages and Systems* 27.3 (May 2005), pp. 477–526. DOI: 10.1145/1065887.1065892 (cit. on pp. 25, 145, 157, 165).

[163] George C. Necula, Scott McPeak and Westley Weimer. 'CCured: Type-Safe Retrofitting of Legacy Code'. In: *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon). New York, NY, USA: ACM, 2002, pp. 128–139. DOI: 10.1145/503272.503286 (cit. on p. 165).

[164] Nicholas Nethercote. *Dynamic Binary Analysis and Instrumentation*. Technical Report UCAM-CL-TR-606. University of Cambridge, Computer Laboratory, 2004. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-606.html (cit. on p. 137).

[165] Nicholas Nethercote and Julian Seward. 'How to Shadow Every Byte of Memory Used by a Program'. In: *Proceedings of the 3rd International Conference on Virtual Execution Environments* (San Diego, California, USA). New York, NY, USA: ACM, 2007, pp. 65–74. DOI: 10.1145/1254810.1254820 (cit. on p. 137).

[166] Nicholas Nethercote and Julian Seward. 'Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation'. In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA). New York, NY, USA: ACM, 2007, pp. 89–100. DOI: 10.1145/1250734.1250746 (cit. on pp. 29, 137, 165).

[167] Kyndylan Nienhuis, Alexandre Joannou, Thomas Bauereiss, Anthony Fox, Michael Roe, Brian Campbell, Matthew Naylor, Robert M. Norton, Simon W. Moore, Peter G. Neumann, Ian Stark, Robert N. M. Watson and Peter Sewell. 'Rigorous Engineering for Hardware Security: Formal Modelling and Proof in the CHERI Design and Implementation Process'. In: *2020 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2020, pp. 1007–1024. DOI: 10.1109/SP40000.2020.00055 (cit. on pp. 29, 40, 94, 103).

[168] NIST. *National Vulnerability Database: CVE-2017-0144*. 16th Mar. 2017. URL: https://nvd.nist.gov/vuln/detail/CVE-2017-0144 (visited on 26/07/2019) (cit. on p. 15).

[169] Ben Niu and Gang Tan. 'Per-Input Control-Flow Integrity'. In: *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA). New York, NY, USA: ACM, 2015, pp. 914–926. DOI: 10.1145/2810103.2813644 (cit. on p. 27).

[170] Robert M. Norton. *Hardware Support for Compartmentalisation*. Technical Report UCAM-CL-TR-887. University of Cambridge, Computer Laboratory, 2016. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-887.html (cit. on pp. 17, 98, 170).

[171] Thomas Nyman, Ghada Dessouky, Shaza Zeitouni, Aaro Lehikoinen, Andrew Paverd, N. Asokan and Ahmad-Reza Sadeghi. 'HardScope: Hardening Embedded Systems Against Data-Oriented Attacks'. In: *Proceedings of the 56th Annual Design Automation Conference 2019* (Las Vegas, NV, USA). New York, NY, USA: ACM, 2019, 63:1–63:6. DOI: 10.1145/3316781.3317836 (cit. on p. 28).

[172] Yutaka Oiwa. 'Implementation of the Memory-safe Full ANSI-C Compiler'. In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland). New York, NY, USA: ACM, 2009, pp. 259–269. DOI: 10.1145/1542476.1542505 (cit. on pp. 29, 145).

[173] Yutaka Oiwa, Tatsurou Sekiguchi, Eijiro Sumii and Akinori Yonezawa. 'Fail-Safe ANSI-C Compiler: An Approach to Making C Programs Secure Progress Report'. In: *Software Security — Theories and Systems*. Ed. by Mitsuhiro Okada, Benjamin C. Pierce, Andre Scedrov, Hideyuki Tokuda and Akinori Yonezawa. Springer Berlin Heidelberg, 2003, pp. 133–153. ISBN: 978-3-540-36532-7 (cit. on pp. 29, 145).

[174] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber and Christof Fetzer. 'Intel MPX Explained: A Cross-Layer Analysis of the Intel MPX System Stack'. In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 2.2 (June 2018), 28:1–28:30. DOI: 10.1145/3224423 (cit. on pp. 16, 112, 130, 139, 140, 143–145, 165, 166).

[175] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber and Christof Fetzer. *Performance Evaluation*. URL: https://intel-mpx.github.io/performance/ (visited on 27/06/2019) (cit. on pp. 144, 165).

[176] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber and Christof Fetzer. *Usability Evaluation*. URL: https://intel-mpx.github.io/usability/ (visited on 30/07/2019) (cit. on p. 143).

[177] Aleph One. 'Smashing The Stack For Fun And Profit'. In: *Phrack Magazine* 7.49 (8th Nov. 1996). URL: http://phrack.org/issues/49/14.html#article (cit. on p. 25).

[178] Harish Patil and Charles Fischer. 'Low-Cost, Concurrent Checking of Pointer and Array Accesses in C Programs'. In: *Software: Practice and Experience* 27.1 (1997), pp. 87–110. DOI: 10.1002/(SICI)1097-024X(199701)27:1<87::AID-SPE78>3.0.CO;2-P (cit. on p. 145).

[179] Bruce Perens. *Efence(3): Electric Fence Malloc Debugger*. 27th Apr. 1993. URL: https://linux.die.net/man/3/efence (visited on 07/10/2019) (cit. on p. 29).

[180] George Pirocanac. *MIPSpro™ 64-Bit Porting and Transition Guide*. manual 007-2391-006. Silicon Graphics, Inc. URL: http://irix7.com/techpubs/007-2391-006.pdf (cit. on pp. 77, 78, 205).

[181] Qualys. *The Stack Clash*. 19th June 2017. URL: https://www.qualys.com/2017/06/19/stack-clash/stack-clash.txt (visited on 27/09/2019) (cit. on p. 29).

[182] Charles Reis and Steven D. Gribble. 'Isolating Web Programs in Modern Browser Architectures'. In: *Proceedings of the 4th ACM European Conference on Computer Systems* (Nuremberg, Germany). New York, NY, USA: ACM, 2009, pp. 219–232. DOI: 10.1145/1519065.1519090 (cit. on p. 170).

[183] Charles Reis, Alexander Moshchuk and Nasko Oskov. 'Site Isolation: Process Separation for Web Sites within the Browser'. In: 28th USENIX Security Symposium (USENIX Security 19). Santa Clara, CA: USENIX Association, 2019 (cit. on p. 170).

[184] Alexander Richardson. *A C Library to Compress/Decompress CHERI Capabilities: CTSRD-CHERI/cheri-compressed-cap*. Capability Hardware Enhanced RISC Instructions, 15th June 2019. URL: https://github.com/CTSRD-CHERI/cheri-compressed-cap (visited on 24/06/2019) (cit. on p. 55).

[185] Alexander Richardson. *Avoid creating an out-of-bounds pointer in word() by arichardson · Pull Request #48 · onetrueawk/awk*. 2nd Sept. 2019. URL: https://github.com/onetrueawk/awk/pull/48 (visited on 02/09/2019) (cit. on p. 136).

[186] Alexander Richardson. *cheribuild*. Capability Hardware Enhanced RISC Instructions. URL: https://github.com/CTSRD-CHERI/cheribuild (visited on 07/08/2019) (cit. on p. 150).

[187] Alexander Richardson. *Some Simple Microbenchmarks for CHERI*. 28th July 2019. URL: https://github.com/arichardson/simple-cheri-benchmarks (visited on 28/07/2019) (cit. on pp. 54, 101).

[188] Alexander Richardson and Robert N. M. Watson. 'Secure Linking in the CheriBSD Operating System'. Principles of Secure Compilation (PriSC) 2019 (Cascais, Portugal). 13th Jan. 2019. URL: https://popl19.sigplan.org/details/prisc-2019/11/Secure-Linking-in-the-CheriBSD-Operating-System (cit. on pp. 20, 74).

[189] Manuel Rigger, Stefan Marr, Stephen Kell, David Leopoldseder and Hanspeter Mössenböck. 'An Analysis of X86-64 Inline Assembly in C Programs'. In: *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Williamsburg, VA, USA). New York, NY, USA: ACM, 2018, pp. 84–99. DOI: 10.1145/3186411.3186418 (cit. on pp. 113, 114).

[190] RISC-V Foundation. *RISC-V ELF psABI Specification*. RISC-V, 23rd July 2019. URL: https://github.com/riscv/riscv-elf-psabi-doc/blob/master/riscv-elf.md (visited on 09/08/2019) (cit. on p. 77).

[191] Ryan Roemer, Erik Buchanan, Hovav Shacham and Stefan Savage. 'Return-Oriented Programming: Systems, Languages, and Applications'. In: *ACM Transactions on Information and System Security* 15.1 (Mar. 2012), 2:1–2:34. DOI: 10.1145/2133375.2133377 (cit. on p. 26).

[192] Anne Rogers, Martin C. Carlisle, John H. Reppy and Laurie J. Hendren. 'Supporting Dynamic Data Structures on Distributed-Memory Machines'. In: *ACM Transactions on Programming Languages and Systems* 17.2 (Mar. 1995), pp. 233–263. DOI: 10.1145/201059.201065 (cit. on p. 159).

[193] Andrew Ruef, Leonidas Lampropoulos, Ian Sweet, David Tarditi and Michael Hicks. 'Achieving Safety Incrementally with Checked C'. In: *Principles of Security and Trust*. Ed. by Flemming Nielson and David Sands. Springer International Publishing, 3rd Apr. 2019, pp. 76–98. ISBN: 978-3-030-17138-4 (cit. on p. 157).

[194] Jerome H. Saltzer. 'Protection and the Control of Information Sharing in Multics'. In: *Commun. ACM* 17.7 (July 1974), pp. 388–402. DOI: 10.1145/361011.361067 (cit. on pp. 74, 94).

[195] Jerome H. Saltzer and Michael D. Schroeder. 'The Protection of Information in Computer Systems'. In: *Proceedings of the IEEE* 63.9 (Sept. 1975), pp. 1278–1308. DOI: 10.1109/PROC.1975.9939 (cit. on pp. 74, 94).

[196] Hiroshi Sasaki, Miguel A. Arroyo, M. Tarek Ibn Ziad, Koustubha Bhat, Kanad Sinha and Simha Sethumadhavan. 'Practical Byte-Granular Memory Blacklisting Using Califorms'. In: (5th June 2019). arXiv: 1906.01838 [cs]. URL: http://arxiv.org/abs/1906.01838 (cit. on p. 29).

[197] Cole Schlesinger, Karthik Pattabiraman, Nikhil Swamy, David Walker and Benjamin Zorn. 'Modular Protections Against Non-Control Data Attacks'. In: *Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium*. Washington, DC, USA: IEEE Computer Society, 2011, pp. 131–145. DOI: 10.1109/CSF.2011.16 (cit. on p. 28).

[198] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi and Thorsten Holz. 'Counterfeit Object-Oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications'. In: *2015 IEEE Symposium on Security and Privacy*. 2015 IEEE Symposium on Security and Privacy. Washington, DC, USA: IEEE Computer Society, 17th–21st May 2015, pp. 745–762. DOI: 10.1109/SP.2015.51 (cit. on pp. 26, 64).

[199] Donn Seeley. *A Tour of the Worm*. UUCS-89-009. University of Utah, Feb. 1989. URL: https://collections.lib.utah.edu/details?id=702918 (cit. on p. 25).

[200] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee and Brad Chen. 'Adapting Software Fault Isolation to Contemporary CPU Architectures'. In: Proceedings of the 19th USENIX Conference on Security. USENIX Association, 11th Aug. 2010. URL: https://www.usenix.org/legacy/events/sec10/tech/full_papers/Sehr.pdf (cit. on p. 29).

[201] Konstantin Serebryany, Derek Bruening, Alexander Potapenko and Dmitry Vyukov. 'AddressSanitizer: A Fast Address Sanity Checker'. In: *USENIX ATC 2012*. 2012. URL: https://www.usenix.org/conference/usenixfederatedconferencesweek/addresssanitizer-fast-address-sanity-checker (cit. on pp. 29, 137, 165, 166).

[202] Kostya Serebryany. 'ARM Memory Tagging Extension and How It Improves C/C++ Memory Safety'. In: *;login: the USENIX Magazine* 44.2 (2019), p. 5. URL: https://www.usenix.org/publications/login/summer2019/serebryany (cit. on pp. 29, 165).

[203] Kostya Serebryany. 'Hardware Memory Tagging to Make C/C++ Memory Safe(r)'. Intel Security Conference (iSecCon) 2018 (Hillsboro, OR, United States.). 4th–5th Dec. 2018 (cit. on p. 15).

[204] Kostya Serebryany. *Simple Guided Fuzzing for Libraries Using LLVM's New libFuzzer*. 9th Apr. 2015. URL: http://blog.llvm.org/2015/04/fuzz-all-clangs.html (visited on 04/09/2019) (cit. on p. 154).

[205] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich and Dmitry Vyukov. 'Memory Tagging and How It Improves C/C++ Memory Safety'. In: (26th Feb. 2018). arXiv: 1802.09517 [cs]. URL: http://arxiv.org/abs/1802.09517 (cit. on pp. 29, 165).

[206] Kostya Serebryany, Evgenii Stepanov and Vlad Tsyrklevich. 'Memory Tagging, How It Improves C++ Memory Safety, and What Does It Mean for Compiler Optimizations'. 2018 Bay Area LLVM Developers' Meeting (San Jose, CA). 17th–18th Oct. 2018. URL: https://llvm.org/devmtg/2018-10/slides/Serebryany-Stepanov-Tsyrklevich-Memory-Tagging-Slides-LLVM-2018.pdf (cit. on pp. 15, 165).

[207] Hovav Shacham. 'The Geometry of Innocent Flesh on the Bone: Return-into-Libc Without Function Calls (on the X86)'. In: *Proceedings of the 14th ACM Conference on Computer and Communications Security* (Alexandria, Virginia, USA). New York, NY, USA: ACM, 2007, pp. 552–561. DOI: 10.1145/1315245.1315313 (cit. on p. 26).

[208] Michael Siff, Satish Chandra, Thomas Ball, Krishna Kunchithapadam and Thomas Reps. 'Coping with Type Casts in C'. In: *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Toulouse, France). Berlin, Heidelberg: Springer-Verlag, 1999, pp. 180–198. DOI: 10.1145/318774.318942 (cit. on p. 129).

[209] Matthew S. Simpson and Rajeev K. Barua. 'MemSafe: Ensuring the Spatial and Temporal Memory Safety of C at Runtime'. In: *Proceedings of the 10th IEEE Working Conference on Source Code Analysis and Manipulation* (Timisoara, Romania). IEEE, Sept. 2010, pp. 199–208. DOI: 10.1109/SCAM.2010.15 (cit. on pp. 29, 111, 137, 142, 165).

[210] Matthew S. Simpson and Rajeev K. Barua. 'MemSafe: Ensuring the Spatial and Temporal Memory Safety of C at Runtime'. In: *Software: Practice and Experience* 43.1 (2nd Feb. 2012), pp. 93–128. DOI: 10.1002/spe.2105 (cit. on p. 142).

[211] Kanad Sinha and Simha Sethumadhavan. 'Practical Memory Safety with REST'. In: *Proceedings of the 45th Annual International Symposium on Computer Architecture* (Los Angeles, California). Piscataway, NJ, USA: IEEE, 2018, pp. 600–611. DOI: 10.1109/ISCA.2018.00056 (cit. on p. 29).

[212] Prokash Sinha. 'A Memory-Efficient Doubly Linked List'. In: *Linux Journal* 2005.129 (Jan. 2005), pp. 10–. URL: http://dl.acm.org/citation.cfm?id=1044989.1044999 (cit. on p. 69).

[213] Lau Skorstengaard, Dominique Devriese and Lars Birkedal. 'Reasoning About a Machine with Local Capabilities'. In: *Programming Languages and Systems*. 27th European Symposium on Programming, ESOP 2018. Ed. by Amal Ahmed. Thessaloniki, Greece: Springer International Publishing, Apr. 2018, pp. 475–501. DOI: 10.1007/978-3-319-89884-1_17 (cit. on p. 105).

[214] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen and A. Sadeghi. 'Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization'. In: *2013 IEEE Symposium on Security and Privacy*. 2013 IEEE Symposium on Security and Privacy. May 2013, pp. 574–588. DOI: 10.1109/SP.2013.45 (cit. on p. 27).

[215] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen and Michael Franz. 'SoK: Sanitizing for Security'. In: (12th June 2018). arXiv: 1806.04355 [cs]. URL: http://arxiv.org/abs/1806.04355 (cit. on pp. 16, 143).

[216] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen and Michael Franz. 'SoK: Sanitizing for Security'. In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019 IEEE Symposium on Security and Privacy (SP) (San Francisco, CA, USA, USA). IEEE, 20th–22nd May 2019, pp. 1275–1295. DOI: 10.1109/SP.2019.00010 (cit. on pp. 140, 165).

[217] Eugene H. Spafford. 'The Internet Worm Program: An Analysis'. In: *SIGCOMM Comput. Commun. Rev.* 19.1 (Jan. 1989), pp. 17–57. DOI: 10.1145/66093.66095 (cit. on p. 25).

[218] SQLite Consortium. *About SQLite*. URL: https://sqlite.org/about.html (visited on 28/07/2019) (cit. on p. 153).

[219] Standard Performance Evaluation Corporation. *Requirements SPEC CPU2006*. 7th Sept. 2011. URL: https://www.spec.org/cpu2006/Docs/system-requirements.html (visited on 19/09/2019) (cit. on p. 161).

[220] Standard Performance Evaluation Corporation. *SPEC CINT2006 Benchmarks*. 24th Aug. 2006. URL: https://www.spec.org/cpu2006/CINT2006/ (visited on 25/06/2019) (cit. on p. 161).

[221] Laszlo Szekeres, Mathias Payer, Tao Wei and Dawn Song. 'SoK: Eternal War in Memory'. In: *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2013, pp. 48–62. DOI: 10.1109/SP.2013.13 (cit. on pp. 16, 17, 25, 27, 29, 74, 158, 167, 169).

[222] Jack Tang. *Exploring Control Flow Guard in Windows 10*. Trend Micro Threat Solution Team, 2015. URL: https://documents.trendmicro.com/assets/wp/exploring-control-flow-guard-in-windows10.pdf (cit. on p. 27).

[223] Ian Lance Taylor. *64-Bit PowerPC ELF Application Binary Interface Supplement 1.9*. 21st July 2004. URL: http://refspecs.linuxfoundation.org/ELF/ppc64/PPC-elf64abi-1.9.pdf (cit. on pp. 77, 81, 90).

[224] Ian Lance Taylor. *Linkers Part 6: Relocations*. 29th Aug. 2007. URL: https://www.airs.com/blog/archives/43 (visited on 09/09/2019) (cit. on p. 33).

[225] The PaX Team. *Address Space Layout Randomization*. 15th Mar. 2003. URL: https://pax.grsecurity.net/docs/aslr.txt (visited on 09/09/2019) (cit. on pp. 16, 26, 89).

[226] The PaX Team. *Non-Executable Pages Design & Implementation*. 1st May 2003. URL: https://pax.grsecurity.net/docs/noexec.txt (visited on 09/09/2019) (cit. on p. 26).

[227] The PaX Team. 'RAP: RIP ROP'. Hackers To Hackers Conference (H2HC) 2015. 24th Oct. 2015. URL: https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf (cit. on p. 76).

[228] The PaX Team. *Re: Vmsplice Exploits, Stack Protector and Makefiles*. E-mail. URL: https://lwn.net/Articles/269532/ (visited on 30/09/2019) (cit. on p. 111).

[229] The PostgreSQL Global Development Group. *PostgreSQL: About*. URL: https://www.postgresql.org/about/ (visited on 27/06/2019) (cit. on p. 152).

[230] The PostgreSQL Global Development Group. *PostgreSQL: Documentation: 9.6: Initdb*. URL: https://www.postgresql.org/docs/9.6/app-initdb.html (visited on 27/06/2019) (cit. on p. 162).

[231] The PostgreSQL Global Development Group. *PostgreSQL: Documentation: 9.6: Pgbench*. URL: https://www.postgresql.org/docs/9.6/pgbench.html (visited on 27/06/2019) (cit. on p. 163).

[232] Sami Tolvanen. *Control Flow Integrity in the Android Kernel*. 10th Oct. 2018. URL: https://android-developers.googleblog.com/2018/10/control-flow-integrity-in-android-kernel.html (visited on 16/08/2019) (cit. on pp. 16, 27).

[233] Paul Turner. *Retpoline: A Software Construct for Preventing Branch-Target-Injection - Google Help*. 2018. URL: https://support.google.com/faqs/answer/7625886 (visited on 26/08/2019) (cit. on pp. 16, 29).

[234] UK Research and Innovation. *ISCF Digital Security by Design Research Projects - Invitation for Proposals*. 26th Sept. 2019. URL: https://epsrc.ukri.org/files/funding/calls/2019/iscf-digital-security-by-design-research-projects-call-document/ (visited on 26/09/2019) (cit. on pp. 18, 171).

[235] Jeffrey Vander Stoep. 'Android: Protecting the Kernel'. Linux Security Summit (Toronto, Ontario, Canada). 26th Aug. 2016. URL: https://events.static.linuxfound.org/sites/events/files/slides/Android-%20protecting%20the%20kernel.pdf (cit. on pp. 15, 17, 25).

[236] Victor van der Veen, Dennis Andriesse, Manolis Stamatogiannakis, Xi Chen, Herbert Bos and Cristiano Giuffrdia. 'The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later'. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security - CCS '17*. The 2017 ACM SIGSAC Conference. Dallas, Texas, USA: ACM, 2017, pp. 1675–1689. DOI: 10.1145/3133956.3134026 (cit. on p. 28).

[237] Vendicator. *Stack Shield*. 8th Jan. 2000. URL: http://www.angelfire.com/sk/stackshield/info.html (visited on 23/08/2019) (cit. on p. 16).

[238] Lluïs Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion and Mateo Valero. 'CODOMs: Protecting Software with Code-Centric Memory Domains'. In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Minneapolis, Minnesota, USA). Piscataway, NJ, USA: IEEE, 2014, pp. 469–480. ISBN: 978-1-4799-4394-4. URL: http://dl.acm.org/citation.cfm?id=2665671.2665741 (cit. on p. 29).

[239] Robert Wahbe, Steven Lucco, Thomas E. Anderson and Susan L. Graham. 'Efficient Software-Based Fault Isolation'. In: *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* (Asheville, North Carolina, USA). New York, NY, USA: ACM, 1993, pp. 203–216. DOI: 10.1145/168619.168635 (cit. on p. 29).

[240] Robert N. M. Watson. *CHERI: The Digital Security by Design (DSbD) Initiative*. 27th Sept. 2019. URL: https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/dsbd.html (visited on 09/10/2019) (cit. on p. 18).

[241] Robert N. M. Watson, Jonathan Anderson, Ben Laurie and Kris Kennaway. 'Capsicum: Practical Capabilities for UNIX'. In: *Proceedings of the 19th USENIX Conference on Security*. Berkeley, CA, USA: USENIX Association, 2010, pp. 3–3. URL: http://dl.acm.org/citation.cfm?id=1929820.1929824 (cit. on p. 29).

[242] Robert N. M. Watson, John Baldwin, David Chisnall, Brooks Davis, Khilan Gudka, Wojciech Koszek, Simon W. Moore, Steven J. Murdoch, Edward Napierala, Peter G. Neumann, Alexander Richardson, Stacey Son, Andrew Turner and Jonathan Woodruff. *Capability Hardware Enhanced RISC Instructions: CHERI Programmer's Guide*. Technical Report (number to be assigned). University of Cambridge, Computer Laboratory variant 1. (to be released) (cit. on p. 38).

[243] Robert N. M. Watson, Brooks Davis and Alexander Richardson. *An Introduction to Pure-Capability CHERI C/C++ Programming*. 10th July 2019 (cit. on p. 38).

[244] Robert N. M. Watson, Simon W. Moore, Peter Sewell and Peter G. Neumann. *An Introduction to CHERI*. Technical Report UCAM-CL-TR-941. University of Cambridge, Computer Laboratory, 2019. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-941.html (cit. on pp. 17, 31, 32).

[245] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Simon W. Moore, Steven J. Murdoch and Michael Roe. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture*. Technical Report UCAM-CL-TR-864. University of Cambridge, Computer Laboratory, 2014. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-864.html (cit. on p. 56).

[246] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, David Chisnall, Brooks Davis, Nathaniel Wesley Filardo, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alex Richardson, Peter Rugg, Peter Sewell, Stacey Son and Hongyan Xia. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 7)*. Technical Report UCAM-CL-TR-927. University of Cambridge, Computer Laboratory, June 2019. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-927.html (cit. on pp. 17, 20, 30, 32, 51, 54, 66, 98, 99, 108, 167, 207).

[247] Robert N. M. Watson, Robert M. Norton, Jonathan Woodruff, Simon W. Moore, Peter G. Neumann, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Michael Roe, Nirav H. Dave, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Ed Maste, Steven J. Murdoch, Colin Rothwell, Stacey D. Son and Munraj Vadera. 'Fast Protection-Domain Crossing in the CHERI Capability-System Architecture'. In: *IEEE Micro* 36.5 (Sept. 2016), pp. 38–49. DOI: 10.1109/MM.2016.84 (cit. on pp. 17, 18, 29, 31, 170).

[248] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son and Munraj Vadera. 'CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization'. In: *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2015, pp. 20–37. DOI: 10.1109/SP.2015.9 (cit. on pp. 17, 18, 29, 31, 80, 170, 215).

[249] Robert N. M. Watson, Jonathan Woodruff, Michael Roe, Simon W. Moore and Peter G. Neumann. *Capability Hardware Enhanced RISC Instructions (CHERI): Notes on the Meltdown and Spectre Attacks*. Technical Report UCAM-CL-TR-916. University of Cambridge, Computer Laboratory, 2018. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-916.html (cit. on pp. 32, 108, 167).

[250] Florian Weimer. *Recommended Compiler and Linker Flags for GCC*. 21st Mar. 2018. URL: https://developers.redhat.com/blog/2018/03/21/compiler-and-linker-flags-gcc/ (visited on 14/08/2019) (cit. on pp. 16, 26).

[251] John Wilander and Mariam Kamkar. 'A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention'. In: *Proceedings of the Network and Distributed System Security Symposium 2003*. 2003. URL: https://www.ndss-symposium.org/ndss2003/comparison-publicly-available-tools-dynamic-buffer-overflow-prevention/ (cit. on p. 76).

[252] John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar and Wouter Joosen. 'RIPE: Runtime Intrusion Prevention Evaluator'. In: *Proceedings of the 27th Annual Computer Security Applications Conference* (Orlando, Florida, USA). New York, NY, USA: ACM, 2011, pp. 41–50. DOI: 10.1145/2076732.2076739 (cit. on pp. 130, 139).

[253] Eric Wimberley. *Bypassing AddressSanitizer*. 5th Sept. 2013, p. 5. URL: https://dl.packetstormsecurity.net/papers/general/BreakingAddressSanitizer.pdf (cit. on pp. 29, 112).

[254] Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Anthony Fox, Robert Norton, Thomas Baureiss, David Chisnall, Brooks Davis, Khilan Gudka, Nathaniel Wesley Filardo, A. Theodore Markettos, Michael Roe, Peter G. Neumann, Robert N. M. Watson and Simon W. Moore. 'CHERI Concentrate: Practical Compressed Capabilities'. In: *IEEE Transactions on Computers* 68.10 (Oct. 2019), pp. 1455–1469. DOI: 10.1109/TC.2019.2914037 (cit. on pp. 17, 30, 38, 44, 51, 52, 55, 98, 164).

[255] Jonathan Woodruff, Alexander Richardson, Nathaniel Wesley Filardo, Robert N. M. Watson, Lucian Paul-Trifu and Simon W. Moore. *Capability Precision Selection Study*. Technical Report (number to be assigned). University of Cambridge, Computer Laboratory. (to be released) (cit. on p. 51).

[256] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton and Michael Roe. 'The CHERI Capability Model: Revisiting RISC in an Age of Risk'. In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Minneapolis, Minnesota, USA). Piscataway, NJ, USA: IEEE, June 2014, pp. 457–468. DOI: 10.1109/ISCA.2014.6853201 (cit. on pp. 16, 17, 31, 38, 58).

[257] Jonathan D. Woodruff. *CHERI: A RISC Capability Machine for Practical Memory Safety*. Technical Report UCAM-CL-TR-858. University of Cambridge, Computer Laboratory, 2014. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-858.html (cit. on p. 58).

[258] Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel W. Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G. Neumann, Simon W. Moore, Robert N. M. Watson and Timothy M. Jones. 'CHERIvoke: Characterising Pointer Revocation Using CHERI Capabilities for Temporal Memory Safety'. In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. Columbus, OH, USA: ACM, 2019, pp. 545–557. DOI: 10.1145/3352460.3358288 (cit. on pp. 17, 21, 30–32, 38, 39, 170, 171).

[259] Hongyan Xia, Jonathan Woodruff, Hadrien Barral, Lawrence Esswood, A. Joannou, Robert Kovacsics, David Chisnall, Micheal Roe, Brooks Davis, Edward Napierala, John Baldwin, Khilan Gudka, Peter G. Neumann, Alexander Richardson, Simon W. Moore and Robert N. M. Watson. 'CheriRTOS: A Capability Model for Embedded Devices'. In: *2018 IEEE 36th International Conference on Computer Design (ICCD)*. 2018 IEEE 36th International Conference on Computer Design (ICCD). Oct. 2018, pp. 92–99. DOI: 10.1109/ICCD.2018.00023 (cit. on pp. 17, 21, 31, 38).

[260] Wei Xu, Daniel C. DuVarney and R. Sekar. 'An Efficient and Backwards-Compatible Transformation to Ensure Memory Safety of C Programs'. In: *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering* (Newport Beach, CA, USA). New York, NY, USA: ACM, 2004, pp. 117–126. DOI: 10.1145/1029894.1029913 (cit. on pp. 29, 160, 166).

[261] Lu Yang and Christopher Barton. *An Overview of the TOC on AIX*. 6th Nov. 2012, p. 10. URL: https://www.ibm.com/developerworks/rational/library/overview-toc-aix/overview-toc-aix-pdf.pdf (cit. on p. 77).

[262] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula and Nicholas Fullagar. 'Native Client: A Sandbox for Portable, Untrusted X86 Native Code'. In: *2009 30th IEEE Symposium on Security and Privacy*. Proceedings of the 2009 30th IEEE Symposium on Security and Privacy. IEEE Computer Society, 17th May 2009, pp. 79–93. DOI: 10.1109/SP.2009.25 (cit. on p. 29).

[263] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R. Sekar, Frank Piessens and Wouter Joosen. 'PAriCheck: An Efficient Pointer Arithmetic Checker for C Programs'. In: *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security* (Beijing, China). New York, NY, USA: ACM, 2010, pp. 145–156. DOI: 10.1145/1755688.1755707 (cit. on pp. 29, 166).

[264] Michael Zhivich and Tim Leek. 'Dynamic Buffer Overflow Detection'. In: *Bugs: Workshop on the Evaluation of Software Defect Detection Tools Evaluation of Software Defect Detection Tools*. 12th June 2005. URL: http://www.cs.umd.edu/~pugh/BugWorkshop05/papers/61-zhivich.pdf (cit. on p. 140).

Acronyms

$ddc	default data capability
$pcc	program counter capability
ABI	application binary interface
API	application programming interface
ASan	AddressSanitizer
ASLR	address-space layout randomization
AST	abstract syntax tree
BTI	ARM Branch Target Indicators
CET	Intel Control-flow Enforcement Technology
CFG	control-flow graph
CFI	Control-flow Integrity
CHERI	Capability Hardware Enhanced RISC Instructions
CheriSH	CHERI sub-object hardening
CPI	Code-pointer Integrity
CPS	Code-pointer Separation
CVE	Common Vulnerabilities and Exposures
CWE	Common Weakness Enumeration
DSO	dynamic shared object
ELF	Executable and Linkable Format, formerly named Extensible Linking Format
FDT	Flattened Device Tree
FPGA	field-programmable gate array
FPU	floating-point unit
GOT	global offset table
GVE	Global Visibility Enforcement
IPC	inter-process communication
IR	intermediate representation
ISA	instruction-set architecture
JIT	just-in-time
KPTI	kernel page-table isolation
LTO	link-time optimization
MMU	memory management unit
MPX	Intel Memory Protection Extensions
OS	operating system
PLT	procedure linkage table
RELRO	read-only after relocation processing

RISC	reduced instruction set computer
ROP	return-oriented programming
RTLD	run-time link-editor or run-time linker
RTTI	run-time type information
SGX	Intel Software Guard Extensions
SLOC	source lines of code
TCB	trusted computing base
TLB	translation lookaside buffer
TLS	thread-local storage
UBSan	UndefinedBehaviorSanitizer

Appendix A

MIPS and CHERI instruction set background

A.1 Calling conventions and register usage

The 64-bit MIPS ISA includes 32 integer registers. Table A.1 lists the assembly-language names, compiler usage and whether the register is caller- or callee-saved in the MIPS n64 ABI [180]. CHERI-MIPS extends the 64-bit MIPS ISA with 32 additional capability registers. The calling convention for pure-capability C/C++ is based on the n64 ABI, but uses capability registers instead of integer registers for e.g. the return and stack pointers. These register conventions are listed in Table A.2. Importantly, both pure-capability linkage models introduced in Chapter 4 — the PC-relative ABI (Section 4.2.4 and the PLT ABI (Section 4.2.5) — use this same calling convention. The only difference between the two ABIs is the usage of `$cgp`. In the PLT ABI, this register holds a pointer to the `captable`, the capability equivalent of a GOT. In the PC-relative ABI, the pointer to the `captable` can be derived from the current program counter, so `$cgp` can be used as an additional temporary register.

Register	ABI Name	Saver	Compiler usage
$0	$zero	Caller	Constant zero register
$1	$at	Caller	Assembler temporary
$2	$v0	Caller	First return value
$3	$v1	Caller	Second return value
$4–$11	$a0–$a7	Caller	Function arguments
$12–$15	$t4–$t7	Caller	Temporaries
$16–$22	$s0–$s6	Callee	Saved registers
$23	$s7	Caller	Base pointer
$24	$t8	Caller	Temporary
$25	$t9	Caller	Call destination/function entry point
$26–$27	$k0–$k1	N/A	Reserved for kernel
$28	$gp	Callee	Globals pointer
$29	$sp	Callee	Stack pointer
$31	$fp	Callee	Frame pointer
$31	$ra	Caller	Return address

Table A.1: MIPS n64 register conventions

Register	ABI Name	Saver	Compiler usage
$0	$zero	Caller	Constant zero register
$1	$at	Caller	Assembler temporary
$2	$v0	Caller	First integer return value
$3	$v1	Caller	Second integer return value
$4–$11	$a0–$a7	Caller	Integer arguments
$1–$15	$t4–$t7	Caller	Integer temporaries
$16–$23	$s0–$s7	Callee	Saved integer registers
$24–$25	$t8–$t9	Caller	Integer temporaries
$26–$27	$k0–$k1	N/A	Reserved for kernel
$28–$30		Callee	Saved integer register
$31		Caller	Integer temporary
$c0	$cnull	Callee	Constant NULL capability register
$c1–$c2		Caller	Capability temporaries
$c3		Caller	Capability return value/first argument
$c4–$c10		Caller	Capability arguments‘
$c11	$csp	Caller	Stack capability
$c12		Caller	Call destination/function entry point
$c13		Caller	Capability to on-stack arguments
$c14–$c15		Caller	Temporary capability registers
$c16		Caller	Exception pointer register
$c17	$cra	Callee	Return capability
$c18–$c23		Callee	Saved capability registers
$c24	$cfp	Callee	Capability frame pointer
$c25	$cbp	Callee	Capability base pointer
$c26	$cgp	Caller	Capability globals pointer
$c27–$c31		Caller	Not used by the compiler

Table A.2: Pure-capability C/C++ register conventions

Mnemonic	Description
CGetAddr	Move capability address to integer register
CGetAndAddr	Move capability address to integer register, with mask
CAndAddr	Mask address of capability
CSetAddr	Set capability address
CGetPCCIncOffset	Move `$pcc` to capability register and increment offset
CGetPCCSetAddr	Move `$pcc` to capability register with new address
CLCBI	Load capability via capability (with bigger immediate)
CSealEntry	Construct a *sentry* capability
CReadHwr	Read a special-purpose capability register
CWriteHwr	Write a special-purpose capability register

Table A.3: New CHERI-MIPS instructions added as part of this dissertation

A.2 List of CHERI instructions

Tables A.3 and A.4 list the CHERI instructions mentioned throughout this dissertation with a brief description of their functionality. For a full list and longer descriptions as well as a Sail [14] specification of each instruction please see the CHERI Architecture Reference [246].

Mnemonic	Description
CGetBase	Move capability base to an integer register
CGetLen	Move capability length to an integer register
CGetOffset	Move capability offset to an integer register
CGetPerm	Move capability permissions to an integer register
CGetSealed	Test if a capability is sealed
CGetTag	Move tag bit to an integer register
CGetType	Move object type to an integer register
CEq	Test if capabilities are equal
CExEq	Test if capabilities are exactly equal (including metadata)
CLT	Test if capability less than
CLTU	Test if capability less than (unsigned)
CLE	Test if capability less or equal than
CLEU	Test if capability less or equal than (unsigned)
CToPtr	Capability to integer pointer
CAndPerm	Restrict capability permissions
CBuildCap	Create a capability from in-memory representation
CClearTag	Clear the tag bit
CFromPtr	Create capability from integer offset
CFromDDC	Create capability from `$ddc`-relative offset
CGetPCC	Move `$pcc` to capability register
CGetPCCSetOffset	Move `$pcc` to capability register with new offset
CIncOffset	Increment capability offset
CIncOffsetImm	Increment capability offset by immediate
CMove	Move capability
CSetBounds	Set bounds (round if not representable)
CSetBoundsImm	Set bounds immediate (round if not representable))
CSetBoundsExact	Set bounds (trap if not representable)
CSetOffset	Set cursor to an offset from base
CSub	Subtract capabilities
CL[BHWD][U]	Load integer via capability
CLC	Load capability via capability register
CLL[BHWD][U]	Load linked integer via capability
CLLC	Load linked capability via capability
CSC	Store capability via capability
CS[BHWD]	Store integer via capability
CSC[BHWD]	Store conditional integer via capability
CSCC	Store conditional capability via capability
CBEZ	Branch if capability is NULL
CBNZ	Branch if capability is not NULL
CBTS	Branch if capability tag is set
CBTU	Branch if capability tag is unset
CCall	Call into another security domain
CJALR	Jump and link capability register, unsealing *sentry* capabilities
CJR	Jump to capability register, unsealing *sentry* capabilities

Table A.4: CHERI-MIPS instruction overview

Appendix B

Indirect `CCall` capabilities

In order to isolate different domains (in this work DSOs), a non-monotonic transition to the other domain is required. Currently, we can transition to another domain (non-monotonically) using `CCall` or *sentry* capabilities (see Section 4.7.1). However, `CCall` requires two capabilities that are linked together by having the same *otype* field and this *otype* field is a limited resource that will need reclaiming eventually. There is also a problem with *sentry* capabilities: to associate data with the *sentry* capability, this data needs to be uniquely identified by the code location. Therefore, each *sentry* capability that needs different data for the same code requires a separate trampoline. This can be a problem if we want to use *sentry* capabilities for return trampolines that restore the stack (see Section 4.8.4.1) as each call would require allocation of a read-write-execute trampoline.

I propose instead to use an *indirect CCall* capability, which is a sealed capability that points to a pair of capabilities. Rather than linking capabilities by *otype*, we could have them linked by virtue of being in adjacent memory locations. This removes the need for return trampolines for stack spills since we can now associate data (the original stack pointer) with the jump target (the return address) and do not have to allocate a unique *otype* for every trust domain. Indirect `CCall` capabilities can use the same encoding scheme as *sentry* capabilities and use a hardware-defined reserved *otype* field. Indirect `CCall` capabilities can enforce compartmentalization as long as no unsealed capability to the indirect location is leaked to the callee.

To use indirect `CCall`, I propose a new instruction `CCallIndirect` (which can just be encoded using a new selector number in the `CCall` encoding space). This instruction takes a single sealed (untyped) capability that points to a memory location containing two capabilities. The first one is a code capability (ideally a *sentry* capability, but that is not necessarily required), immediately followed by another capability. The first capability is installed into `$pcc`, and the second one into `$idc` (which in the current ABIs is equal to `$cgp`).

We have not yet implemented this instruction, however it should be possible to implement for MIPS, ARM and possibly RISC-V. MIPS has branch delay slots and it

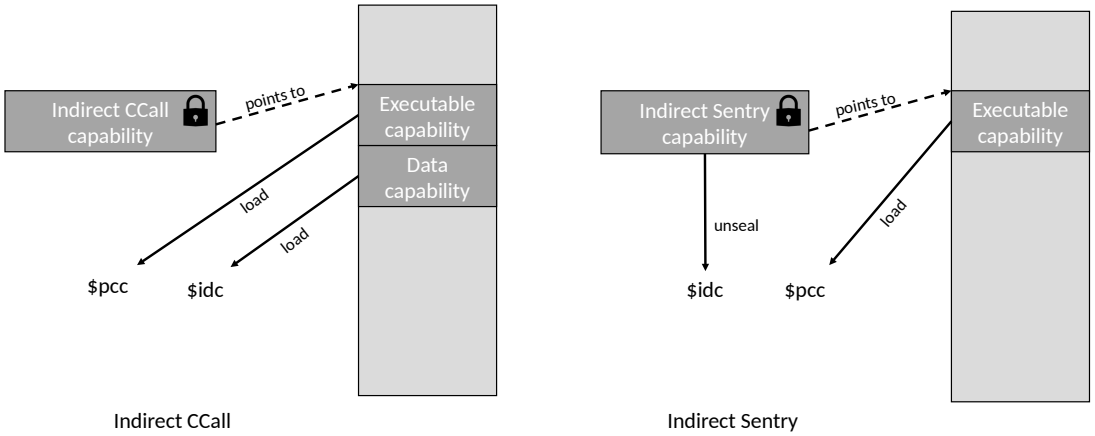


Figure B.1: The two variants of indirect CCall.

could be possible to reuse the branch delay slot logic to perform the second data load.[1] Furthermore, ARM (AArch32) already has a load into `$pc` instruction and a load-pair instruction. Therefore, it should be possible to implement indirect `CCall` (as a multi-cycle instruction) since it is similar to a load-pair instruction with one of the destinations being `$pcc`.

B.1 Replacing PLT stubs with indirect `CCall`

In addition to being able to use indirect `CCall` for stack isolation (see Section 4.8.4.1), it could also be used to elide the code part of the PLT stubs used for domain transition since it performs exactly the same steps as the current PLT stubs. However, to be able to use indirect `CCall` for all call sites, the compiler would have to use indirect `CCall` for all (potentially) external calls. For calls within the same DSO, we would have to add the target `$cgp` next to every call target in the `captable`. This would increase start-up time since more capabilities need to be initialized.[2] Alternatively, it might be possible to adjust indirect `CCall` to behave as if it were a jump if the data argument is unsealed. This would allow it to be used for all call-sites. However, it is unclear if this could be implemented in a complex pipeline, where data-dependent behaviour changes come at a significant cost.

B.2 A simpler indirect `CCall` (indirect *sentries*)

Jonathan Woodruff proposed a variant of the indirect `CCall` that should be easier to implement for simpler CPU architectures while retaining most of the expressiveness. In

[1] This approach could be used in the current FPGA implementation as it allows `$pcc` to be the destination of a load (although no instruction uses this). Moreover, even in cases where the load in the delay slot trapped, no architectural state would leak to the exception handler since the exception `$pcc` (`$epcc`) register would hold the address of the indirect `CCall` and not the newly loaded value.

[2] In the PC-relative ABI we could use a `NULL` value, since `$cgp` is overwritten by the callee. This would avoid the start-up overhead, but still unnecessarily grow the `captable`.

this scheme, the indirect `CCall` does not load a data capability as the second step, but instead places an unsealed version of the indirect `CCall` capability in register `$idc`. Unlike the full indirect `CCall`, this grants access to the memory location used to store the code (and data) pointers to the callee. For the use-cases listed above this could work but may require additional design changes. If we are using indirect `CCall` to limit the stack, then the target of the indirect `CCall` must have allocated the memory region for the pair of capabilities and therefore already has access. In case we are using it for PLT stubs allocated by RTLD, we would have to ensure that the unsealed capability is read-only as the callee could otherwise modify the call destination for following calls. Additionally, if the simplified indirect `CCall` were to use the whole `captable` with the offset pointing to the target function, the callee would gain access to all the code capabilities in that table and could invoke them directly without setting `$idc`. Therefore, I believe the simplified indirect `CCall` would work very well for securing the stack and return address but is not an ideal match for replacing PLT stubs unless this approach also sets the offset of the unsealed capability to zero.

Appendix C

Initializing global capabilities

In the following subsections I explain how the `__cap_relocs` mechanism works, why it is not ideal for dynamically linked code, and how I initially worked around these shortcomings until I replaced some uses with ELF relocations. Finally, I propose a better replacement for `__cap_relocs` that will reduce memory overhead and potentially reduce start-up time.

C.1 Processing `__cap_relocs`

The `__cap_relocs` section is an array of structures with 5 elements: the destination location of the relocation, the target address, size, offset and permissions. Each value is a 64-bit integer, so each capability that needs initializing occupies 48 bytes plus the size of a capability (16 or 32 bytes) in the resulting binary (on disk and in-memory). This is a rather large overhead, but Appendix C.4 shows how this overhead can be reduced. Listing C.1 shows a slightly simplified implementation[1] of the start-up capability relocation processing code.

For each relocation entry we first determine if it is a code or a data value and select the base capability to derive the value from. Then we add the target address to the capability, set the bounds for data values,[2] increment the offset to the desired value and finally store the computed value to the desired capability location. This approach is simple and except for the large memory overhead (and thereby increased cache-misses) it is also reasonably fast since it is a loop with a short instruction body and few branches. However, this technique is not sufficient for dynamic linking (as will be explained in the following subsections), and it also results in some non-obvious performance issues.

Another important consideration is that the function `crt_init_globals()` must be invoked before any other function can be called since function calls depend on being able to load a valid target code capability. The function is therefore marked as `always_inline` to ensure that it is part of the initial entry point to the binary and is run prior to any other function calls.

[1] The real implementation also uses another base capability for read-only data in addition to code and data capabilities. Furthermore, it sets bounds on functions when this is safe in the chosen ABI. In the case of dynamically linked code it also adds the DSO base address to `reloc->object` and `reloc->capability_location`.

[2] In the PLT ABI we can also set bounds for code pointers, but in the PC-relative ABI we must use the full DSO `$pcc` value.

```
struct capreloc {
    uint64_t capability_location;
    uint64_t object;
    uint64_t offset;
    uint64_t size;
    uint64_t permissions;
};

__attribute((always_inline)) void
crt_init_globals(void*__capability code_ptr, void* __capability data_ptr) {
    for (struct capreloc *reloc = &__start___cap_relocs; reloc < &__stop___cap_relocs; reloc++) {
        _Bool isFunction = (reloc->permissions & function_flag) == function_flag;
        void **dest = __builtin_cheri_offset_set(data_ptr, reloc->capability_location);
        void *base = isFunction ? code_ptr : data_ptr; // select RX or RW permissions
        void *src = __builtin_cheri_offset_set(base, reloc->object);
        if (!isFunction) // do not bound functions
            src = __builtin_cheri_bounds_set(src, reloc->size);
        src = __builtin_cheri_offset_increment(src, reloc->offset);
        *dest = src;
    }
}
```

Listing C.1: Simplified `__cap_relocs` processing implementation

C.2 Evolving `__cap_relocs` for dynamic linking

Prior to my work we were using `__cap_relocs` to initialize all global capabilities. This works reasonably well for statically linked binaries but, as it was designed as a workaround for lack of CHERI linker support, it has some shortcomings when used for dynamically linked binaries.

C.2.1 Adding correct size information

Due to the use of an external tool, capsizefix, to generated the `__cap_relocs` there was no way to emit correct size information for unnamed constants (such as strings) in C. This is caused by the fact that symbol information for string constants and other compiler-generated symbols, whose names generally start with `.L.<name>` (e.g. `.L.str.1` for the first unnamed string constant), are never included in the linker-generated output file. Originally, the compiler would emit a `__cap_relocs` entry with two `R_MIPS_64` relocations. The first relocation was against the capability location (a relocation against the section plus offset) and the second against the symbol. However, for anonymous symbols (such as for example string constants that end up in `.rodata.str.n`) the target symbol relocation would end up being a relocation against the section plus an offset. Therefore, capsizefix could not emit correct bounds for these symbols and all string constants were bounded to the containing section instead (as this is the strictest bounds that capsizefix can infer).

I fixed this issue by moving the finalizing of the `__cap_relocs` from the external tool capsizefix to the static linker LLD. As LLD sees all the local symbols, it can emit the correct relocation information. I changed the compiler to emit a single `R_CHERI_CAPABILITY` relocation against the target symbol. I further modified LLD to generate the entire

`__cap_relocs` entry based on the `R_CHERI_CAPABILITY` relocation. These changes ensure that string constants have the bounds only of the individual string (since the compiler now emits a local symbol for the string) rather than the whole input section.

There is one more issue related to sizes: the size of a symbol may not be known at static link time since the static linker allows unresolved symbols when building shared libraries.[3] This means that in those cases we could not fill in the size field of the `__cap_relocs` correctly. Instead, we would fill the field with a `-1` constant, which indicated to RTLD to not set bounds on that symbol. In addition, the value of the size field could be wrong if the size of a symbol is different in the library at run time compared to static link time. This mismatch can happen when new fields are added to an exported struct, but we still set the bounds to be the value from static link time (because the binary hasn't been relinked after the shared library change) and therefore potentially get a run-time crash.

This problem can be fixed by emitting a dynamic `R_CHERI_SIZE64` relocation in that field that then gets processed by the dynamic linker. Unlike other architectures such as x86 (`R_X86_64_SIZE32` and `R_X86_64_SIZE64`) MIPS does not have a size relocation. Therefore, I added a `R_MIPS_CHERI_SIZE` relocation to emit the size of a symbol and added support for that relocation to both LLD and the CheriBSD RTLD.[4]

To get the correct size in `__cap_relocs`, LLD now emits a `R_MIPS_CHERI_SIZE | (R_MIPS_64 << 64)` relocation for the length field. This is required for any external and also for local but preemptible symbols (i.e. those which can be overridden at run time by a library that is loaded earlier — e.g. using the `LD_PRELOAD` mechanism). Unfortunately, the default linkage behaviour is to make every symbol preemptible. In the common case where code is not linked with `-Bsymbolic/-Bsymbolic-functions`, this causes many dynamic relocations. Additionally, we already have two dynamic relocations for every `__cap_relocs` entry (one for the location and one for the target) which makes this relocation mechanism very inefficient. In Appendix C.3 I show how this overhead can be reduced.

C.2.2 Reducing privilege during `__cap_relocs` processing

Initially, the `crt_init_globals()` function was rather simple and just looped over everything in the section and set bounds according to that information. This worked fine with a global `$pcc` and `$ddc` but once we started restricting `$pcc` and set `$ddc` to `NULL` we had to change this approach slightly.

We also discovered that this mechanism allowed creation of writable capabilities to the text segment if a programmer forgot to mark a symbol as a function. In that case the `__cap_relocs` code would attempt to populate the target from the global data capability instead of the current `$pcc`. It would then also set tight bounds on the function

[3] This is allowed by default and is only an error when linking with the `-Wl,-z,defs` flag

[4] For consistency with other MIPS relocations `R_MIPS_CHERI_SIZE` is used for a 32-bit value and `R_MIPS_CHERI_SIZE | (R_MIPS_64 << 64)` is used for 64-bit values.

even in the PC-relative ABI and then result in a crash when called because it could not derive `$cgp` from a tightly bounded `$pcc`. This problem was discovered because we had an assembly function `cheri_invoke()` (which is used by the libcheri CCall calling convention [248]) that was missing the function type information in the assembly code (`.type,@function`). This then resulted in a read-write (but not executable) tightly bounded capability to `cheri_invoke()` being placed in the `captable`. Code calling `cheri_invoke()` would then crash due to the capability not having execute permission. This could have theoretically resulted in code being able to write to the text segment. However, we are still using the MMU, so this would have been prevented.

To fix the use of incorrect base capabilities and bring the CHERI capability permissions in line with the MMU permissions, I modified the C start-up code to parse the ELF program headers of the binary. These headers allow us to find the ranges of the code and data segments and create appropriate capabilities instead of passing the full DSO `AT_PHDR` capability to `crt_init_globals()`. The root data capability now excludes the text segment and will therefore result in a trap when attempting to derive code capability from the root data capability.[5] However, we were still using a read-write capability to initialize globals that reside in the read-only data segment since there was no way to distinguish read-only and read-write data in the `__cap_relocs`. I added a new `Constant` flag to the `__cap_relocs` which is now used by `crt_init_globals()` to derive constant pointers from a capability that spans only the read-only segment. In order to avoid surprising run-time crashes, I also added a linker warning to LLD whenever a call relocation is used but the target symbol is not marked as being a function.[6]

After these changes, it is no longer possible to accidentally obtain a read-write capability to code. However, we do not attempt to protect against a maliciously modified binary (e.g. one that has all program headers set to read-write-execute). Nevertheless, this was not a security vulnerability, since even a malicious DSO cannot obtain capabilities to another DSO except for the explicitly exported symbols.

C.2.3 Moving `__cap_relocs` processing to RTLD

To allow using read-only and RELRO sections, I had to change the way that the existing `__cap_relocs` is processed. Previously every shared library was responsible for adding tags to all its capabilities. This was done by passing `-Wl,-init=crt_init_globals` to the static linker. This flag causes a `DT_INIT` entry to be added to the `.dynamic` section. At run-time, RTLD will then invoke the function that is referenced by the `DT_INIT` tag—in this case the `crt_init_globals()` function. However, this `DT_INIT` function runs after the dynamic linker has marked RELRO sections as read-only, so we had to keep

[5] `CSetBounds` is used to bound the resulting capability and it traps when used on capabilities that are out-of-bounds.

[6] This is a warning and not an error since it might be the correct behaviour in some rare cases. However, CheriBSD builds with all linker warnings emitted as errors, so we can prevent these crashes.

any section that contains a global capability variable (even if it is only a `const` pointer) read-write. I fixed this issue by removing the `DT_INIT` function from each shared library and instead processing `__cap_relocs` inside RTLD. The `__cap_relocs` processing inside RTLD runs before marking sections as RELRO, which means that `const` pointers now reside in read-only memory.

C.2.4 Partially replacing the `__cap_relocs` mechanism

Even though it was possible to correctly initialize globals after the changes I made to `__cap_relocs`, we were incurring high overheads for every global: each `__cap_relocs` entry contained three dynamic relocations (two of which require a symbol lookup) before the run-time linker could write the resulting value. As a result, I modified LLD and RTLD to use ELF relocations for all preemptible symbols. These new ELF relocations are `R_MIPS_CHERI_CAPABILITY` (for data symbols and function pointers) and `R_MIPS_CHERI_CAPABILITY_CALL` (for functions called via the PLT).

To set the correct bounds for global variables, RTLD parses the ELF symbol table and uses the `st_size` field in the `Elf_Sym` structure as the size of the global variable. The `st_size` value will correspond to the size of the C/C++ declaration or the size of all instructions for functions. When switching from `__cap_relocs` to this mechanism, I noticed that certain symbols did not have size information. This includes all linker-synthesized variables, such as `_DYNAMIC` (which points to the start of the `.dynamic` section) and `__start_<section>`/`__stop_<section>` symbols that are expected to be added for all sections whose name is a valid C identifier. During relocation processing these variables would then be initialized to a zero-length capability and cause a trap on first dereference.[7] I have since updated LLD to use the size of the section as the `st_size` value and this has allowed us to find a global variable overflow in libFuzzer (see Section 6.1.4).

Even after the addition of new CHERI ELF relocations, we still use `__cap_relocs` for DSO-local symbols. The reason for this is that local symbols are not added to the dynamic symbol table, so we are not able to look up the corresponding values.[8]

C.3 Optimizing `__cap_relocs` processing

As noted earlier, `__cap_relocs` processing in a dynamic binary is unnecessarily slow due to the required dynamic relocations. Since we are still using `__cap_relocs` for local symbols even after adding ELF relocations, it makes sense to improve the performance.

[7] This resulted in crashes when processing C++ constructors using the `__ctors_start` symbol.

[8] We could add local symbols to the dynamic symbol table, but that would increase the size of the binary and result in unnecessary hash table lookups even though all values required for relocation are link-time constants.

C.3.1 Removing unnecessary dynamic relocations

The `__cap_relocs` was originally processed by a start-up function in each DSO. This meant that all values inside the `__cap_relocs` had to be absolute values containing the correct location as there is no reliable way to obtain the load address inside this initialization function without relying on assumptions about the object layout. It also means that there can be up to three relocations per `__cap_relocs` entry: one for the location of the capability, one for the target and one for the size of the target. I was able to remove the relocation for the size by only using `__cap_relocs` globals initialization for DSO-local capabilities and relying on `R_MIPS_CHERI_CAPABILITY` ELF relocations for all external symbols. As the size of local symbols is known at static link time, we no longer need relocations for the size.

A significant portion of the `__cap_relocs` overhead was caused by the two dynamic relocations for every entry. However, after starting to use `R_MIPS_CHERI_CAPABILITY` ELF relocations for all preemptible symbols, we only use the `__cap_relocs` for local symbols such as static variables and function addresses within the same DSO. Therefore, we know that both the location and target must be a fixed offset from the load address. By moving the `__cap_relocs` processing to RTLD it now becomes possible to remove the two remaining relocations since RTLD knows the correct load address.

This allowed me to change the static linker to no longer emit the `R_MIPS_64` relocations for the `location` and `base` fields and indicate to the run-time linker that all `__cap_relocs` are relative.[9] If RTLD encounters this flag, it will add the base address of the current DSO[10] to the relocation location and the desired base value before applying the relocation.

Prior to these changes, there was a significant load-time overhead when running CheriABI binaries (see Figure C.1). After applying this simple optimization, the start-up time of RTLD was reduced to 19.9% of the original time. This means RTLD was spending over 80% of its start-up time processing `__cap_relocs`. This optimization also noticeably reduces the start-up time for many binaries. The first example is /usr/bin/true, a very simple binary that only links against libc. In this case, the removal of the two `R_MIPS_64` relocations per `__cap_relocs` entry resulted in a load time of 60.9% of the original time. For the binary in the CheriBSD base system that links against the most shared libraries, /usr/sbin/sshd (23 shared libraries), this optimization reduces the load time to 84.6% of the original time. The absolute number of instructions during sshd start-up was reduced from 20.7 to 17.5 million, corresponding to a removal of over 3 million unnecessary instructions.

[9]The static linker adds a `DF_MIPS_CHERI_RELATIVE_CAPRELOCS` flag to the `DT_MIPS_CHERI_FLAGS` `.dynamic` table entry to indicate that all `__cap_relocs` entries are relative. If the flag is missing the dynamic linker assumes that all entries in the `__cap_relocs` are absolute addresses. In this way we retained compatibility with binaries created by the old toolchain and avoided a disruptive flag day.

[10]This also includes RTLD itself.

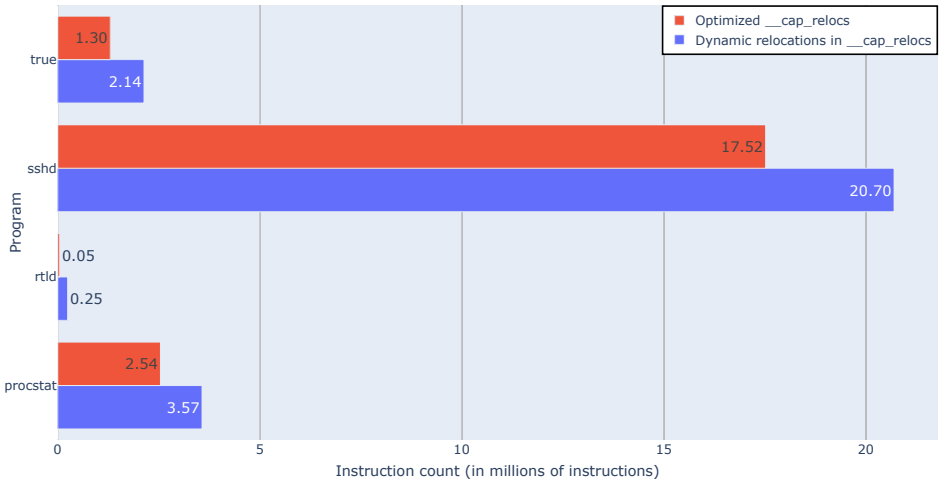


Figure C.1: Start-up time (invoking the `--help` option) for various binaries comparing `__cap_relocs` with and without dynamic `R_MIPS_64` relocations.

C.3.2 Improving efficiency using CBuildCap

The fastest initialization mechanism would use the instruction `CBuildCap` to add a tag to an in-memory representation of capabilities. To do so, `CBuildCap` requires an authorizing capability that is a superset of the raw bits when interpreted as a capability.

While this instruction was originally designed to improve the performance of the kernel swapping out existing pages and reconstructing the capabilities, it is also a near perfect match for the start-up code in both static and dynamic linkage. For statically linked binaries, we could instruct LLD to write the raw capability bits to the destination of the capability relocation and then use `CBuildCap` at run time. A similar approach could be used for dynamic binaries, but we would need a new instruction that can be used to add the DSO load address to these untagged raw bits.

While this approach is the fastest conceivable relocation mechanism, it does have disadvantages. Using `CBuildCap`, the in-memory format of the capability relocation metadata must be identical to the in-memory representation of capabilities, and therefore this format becomes embedded in the binary. This means that the compiler and linker need to be aware of the format and be able to encode it.[11] The bigger problem is that the capability format effectively becomes part of the ABI. Therefore, any future changes to the encoding scheme would mean that binaries compiled against the old format no longer work. These changes include increasing or reducing the number of bits used for bounds precision. Interestingly, such changes already happened twice during my PhD (we changed the bounds precision from 45 to 27 bits and introduced a *flags* field) and may happen again.

[11] This only adds a small amount of logic to the compiler/linker since they already need to be aware of precision constraints imposed by capability compression.

C.4 Proposed mechanism for local symbols

For all local capabilities that need to be initialized I propose to instead embed the necessary relocation information at the location of the target capability (similar to the CBuildCap approach) instead of in a separate section. Using this approach, we no longer add the unnecessarily large `__cap_relocs` to the binary. For example, a statically linked QtWebkit DumpRenderTree binary contains a `__cap_relocs` section with 316008 entries, i.e. a total of 12MiB. If we instead store just a 32-bit offset indicating the relative location of the capability and store the metadata in-place (at the location of the 128-bit or 256-bit capability that is to be initialized), we can reduce the size of the added metadata to 1MB. It is to be noted that this information would also be required for a plain MIPS binary if it is compiled as a position-independent binary (as is increasingly the default). With this proposed scheme, we still encode the same information as the `__cap_relocs` or `CBuildCap` approach but use the following structure[12] instead:

```
union simple_cap_relocation {
    struct {
        uint32_t unused; // only for debugging (check that the right base cap was used)
        uint32_t base;   // byte offset relative to base capability
        uint32_t length; // requested size of capability
        int32_t offset;  // requested capability offset value
    } info;
    void* __capability capability; // resulting relocated value
};
```

In order to initialize a DSO-local capability we now only need a sequence of seven instructions for each relocation: load base, `CIncOffset`, load length, `CSetBounds`, load offset, `CIncOffset`, store capability.[13] One more optimization is possible since the offset will almost always be zero.[14] Therefore, we could list capability relocations that require a non-zero offset separately and shorten the initialization sequence by two instructions for all others.

C.4.1 Restrictions

While this approach has some advantages and we are planning to make it the default mechanism used in CheriBSD, there are some downsides, but we do not consider these critical.

Size limitation This approach only works with 32-bit sizes, i.e. objects greater than 4GB are not supported. As this mechanism is only used for local symbols within a shared

[12] The same structure would also work for CHERI256, if we leave the last 128-bits of the structure uninitialized.

[13] Ideally, this sequence should perform the loads up-front so that they are available in the pipeline by the time they need to be used (since the CHERI FPGA is an in-order CPU).

[14] Yet unlike the `__cap_relocs` approach, we no longer waste space since we are using the target memory location anyway.

object this seems a reasonable restriction. If a need for larger objects is discovered in the future we could replace this with an alternative relocation metadata encoding. This encoding could use a bounds-compression mechanism like CHERI, but with a fixed precision, thereby not exposing the internal representation of capabilities.

No support for CHERI-64 An implementation of CHERI with 64-bit capabilities could not use this relocation data structure as it is too large to fit in-place. We would have to either use a structure that is closer to the in-memory representation of capabilities or store the remaining metadata out-of-band. However, it is clearly possible that CHERI-64 implementations are designed for embedded use-cases and therefore do not require the long-term binary compatibility that a 64-bit general purpose architecture requires. Therefore, embedding the raw capability encoding in memory might be a worthwhile trade-off as it allows reducing the complexity of the CPU implementation and reducing the complexity of the loader code.

C.4.2 Optimizing global capability initialization

To further optimize the process of initializing global capabilities, we could define a new instruction that converts an in-memory *stable* serialized capability format to the current format (e.g. `CTranslateFormat`). This untagged value could then be relocated by adding the relocation base with `CIncOffset`. This works if the added value is sufficiently aligned because the bounds and cursor will stay the same as long the added value is greater than the representable range of the capability. The resulting raw bit representation can then be turned into a valid capability using `CBuildCap`. This approach would reduce the number of instructions in the relocation loop to five: `CLC` to load the bits, `CTranslateFormat`, `CIncOffset`, `CBuildCap` and `CSC`. Moreover, it might be possible to combine the three instructions between the load and the store into only two or even one new instruction by exploiting alignment constraints. However, that would not be as simple to implement in the hardware and can be done in a future revision of the ISA.

Appendix D

Adjusting alignment of variables and types

Prior to the C11 standard, the only way to specify alignment requirements on a variable (such as cache-line alignment for multithreaded code) was to use compiler-specific mechanisms. The GCC solution to this problem is `__attribute__((aligned))`, which according to the documentation 'specifies a minimum alignment (in bytes) for variables of the specified type' [79]. It can also be applied to a structure type in which case it sets the minimum alignment for the entire structure. This is equivalent to specifying it on one of the fields as the structure alignment is the maximum of the alignment of all fields.

However, the documentation for this feature is incorrect. The following comment in the Clang source code explains why:

```
// If the typedef has an aligned attribute on it, it overrides any computed
// alignment we have.  This violates the GCC documentation (which says that
// attribute(aligned) can only round up) but matches its implementation.
```

Due to this inconsistency, changing the location of the attribute results in completely different semantics for almost identical looking statements. Most programmers will be unaware of this subtlety and assume that the GCC documentation is correct. The only reason I discovered this inconsistency is because I was attempting to write tests for new alignment warnings for structures containing capabilities (see below) and was surprised by the IR that Clang was generating for these examples.

```
typedef struct { int *__capability i; } __attribute__((__aligned__(2))) with_cap_td;
with_cap_td var1; // aligned to CAP_SIZE
typedef __attribute__((__aligned__(2))) struct { int *__capability i; } with_cap_td2;
with_cap_td var2; // !Only aligned to 2 bytes!

// Here the attribute can only increase alignment
struct { int *__capability i; } __attribute__((__aligned__(2))) var3;
// ! But here it actually reduces alignment to 2 bytes !
__attribute__((__aligned__(2))) struct { int *__capability i; } var4;
```

The C11 `_Alignas()` alignment specifier behaves more consistently and results in a compiler error when given an alignment that is smaller than the minimum alignment of the annotated type. This makes much more sense than the `__attribute__((aligned))` since it is always possible to get less than the required alignment by using a `char` array or packed union and casting to the desired type.